

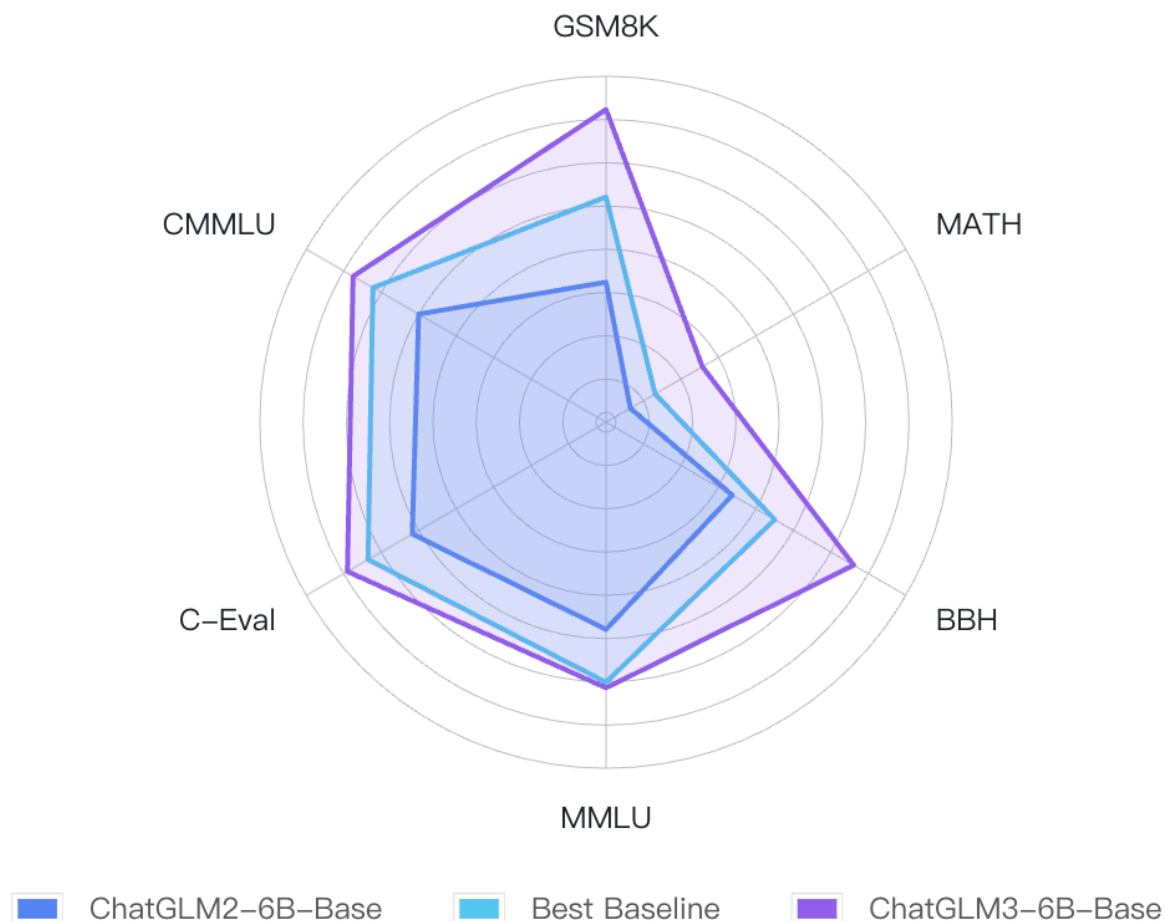
ChatGLM3-6B 开发分享

智谱AI 清华大学 KEG 实验室

<https://github.com/THUDM/ChatGLM3>

更强大的基座

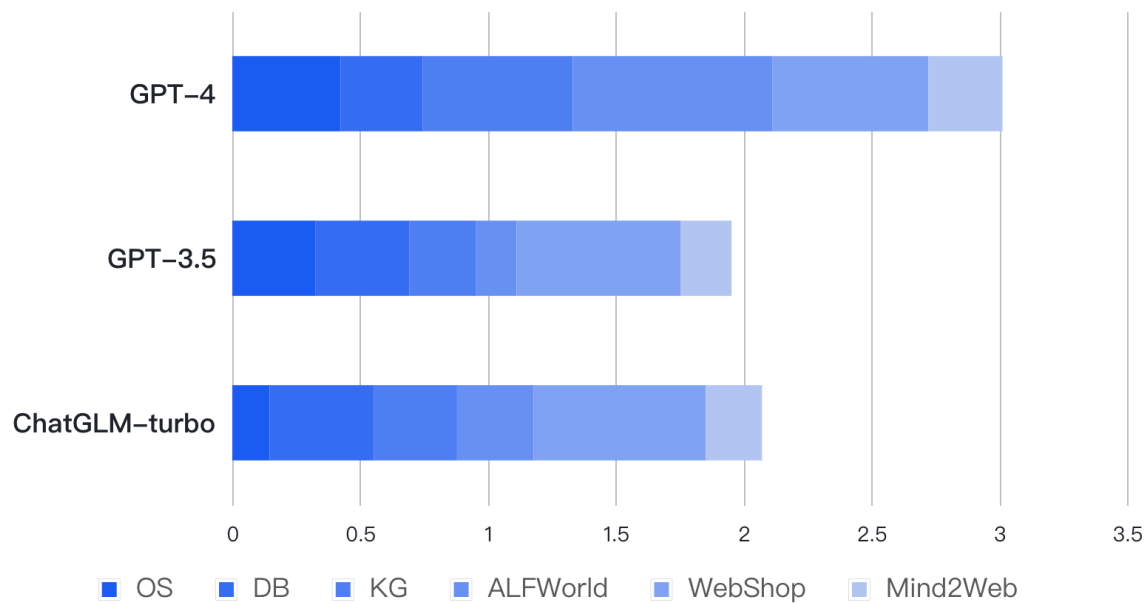
From ChatGLM2 To ChatGLM3



- 训练数据更多样、训练步数更充分、训练策略更合理
- 在语义、数学、推理、代码、知识等不同角度的数据集上表现出色
- 44个中英文公开数据集测试国内第一

更强大的功能

Agent 能力



- 全新设计的 Prompt 格式，在不影响模型通用能力的情况下，全方面增强 chatglm3-6b能力
- 原生支持工具调用 (Function Call)、代码执行 (Code Interpreter) 以及 Agent 任务

Demo 安装与演示

- Steamlit
 - 模型部署为网络应用
- 部署与 Demo 展示
 - 对话能力
 - 工具调用能力
 - 代码解释器能力



LangChain 工具接入：搜索工具

工具描述

```
<System>
...following tools:
[ {...}, ..., {...} ]
```

用户询问

```
<user>
今天北京天气怎么样?
<assistant>
```

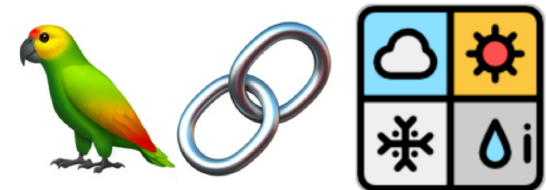
ChatGLM3-6B

```
{
  'role': 'assistant',
  'metadata': 'weather',
  'content': "```python
tool_call(city='Beijing')
```"
}
```

Re-format

```
{
 "action": "weather",
 "action_input": "Beijing"
}
```

Tool Call



Result

# 对话格式

- 角色 special token:

- `<|system|>`

系统提示词, 指明模型扮演的角色等信息

- `<|user|>`

用户输入

- `<|assistant|>`

模型回复

- `<|observation|>`

工具调用、代码执行结果

# 对话格式

- `<|assistant|>{metadata}`:
  - 工具调用: `{metadata}` 为调用的工具名
    - 例如 `<|assistant|>test_tool` 表示模型希望调用 `test_tool` 工具
  - Code Interpreter: 固定为 `<|assistant|>interpreter`

# 对话格式

- 分隔符：角色标签（包括 `metadata`）后跟换行符
  - 标签前不跟
- 示例：
  - `<|system|>`
  - `You are ChatGLM3.<|user|>`
  - `Hi<|assistant|>`
  - `Hello, how can I help you?`



# 对话格式：工具调用

- 固定 system prompt 以发挥最好效果

```
1. <|system|>
2. Answer the following questions as best as you can. You have access to the following tools:
3. [
4. {
5. "name": "track",
6. "description": "追踪指定股票的实时价格",
7. "parameters": {
8. "type": "object",
9. "properties": {
10. "symbol": {
11. "description": "需要追踪的股票代码"
12. }
13. },
14. "required": ['symbol']
15. }
16. }
17.]
```

# 对话格式：工具调用

- 工具定义：JSON 对象列表，三个字段 `name`, `description` 和 `parameters` 分别为工具名、描述、参数定义
- 参数定义可为 JSON Schema 格式

```
1. [
2. {
3. "name": "track",
4. "description": "追踪指定股票的实时价格",
5. "parameters": {
6. "type": "object",
7. "properties": {
8. "symbol": {
9. "description": "需要追踪的股票代码"
10. }
11. },
12. "required": ['symbol']
13. }
14. }
15.]
```

# 对话格式：工具调用

- demo 中的手动模式通过 YAML 定义工具
- tool\_registry 模式在 tool\_registry.py 中定义工具：

```
1. @register_tool
2. def get_weather(
3. city_name: Annotated[str, 'The name of the city to be queried', True],
4.) -> str:
5. """
6. Get the weather for `city_name` in the following week
7. """
8. ...
```

- Annotated 的三个部分分别为参数类型、描述与是否必需
- docstring 会被作为工具的描述

# 对话格式：工具调用

Manual mode ?

Tools

Define your tools in YAML format here:

```
- name: get_current_weather
 description: Get the current weather in a given location
 parameters:
 type: object
 properties:
 location:
 type: string
 description: The city and state, e.g. San Francisco, CA
 unit:
 type: string
 enum:
 - celsius
 - fahrenheit
 required:
 - location
```



今天北京天气怎么样



Calling tool `get_current_weather`:

```
tool_call(location='北京', unit='celsius')
```

Manual mode ?

Tools

Define your tools in YAML format here:

```
- name: bing_search
 description: Execute a search query using Bing Search,
 especially useful for answering questions regarding current
 events.
 parameters:
 type: object
 properties:
 query:
 type: string
 description: The search query, e.g. "Current COVID-19
 cases in the US".
 required:
 - query
```



2023年诺贝尔物理学奖有哪些?



我需要调用搜索引擎来获取2023年诺贝尔物理学奖的相关信息。



Calling tool `bing_search`:

```
tool_call(查询='2023年诺贝尔物理学奖是谁获得')
```

# 对话格式：工具调用输出格式

- 除了正常使用 `<|assistant|>` 进行对话外，模型在感到需要调用工具时会生成 `<|assistant|>{metadata}`，其中 `metadata` 为调用的工具名
- 对话内容为 Python 代码，最终调用 `tool_call` 函数传参数
- 例如：（增加了额外换行以提升可读性）
  - `<|user|>`
  - Can you tell me how the weather is in Beijing now?
  - `<|assistant|>`
  - Sure! I can help with that by querying a weather API.
  - `<|assistant|>get_weather`
  - ````python`
  - `tool_call(location='Beijing')`
  - `````

# 对话格式：工具调用

- 工具调用结果通过 `<|observation|>` 返回给模型

- 例如：

1. `<|user|>`
2. `Can you tell me how the weather is in Beijing now?<|assistant|>`
3. `Sure! I can help with that by querying a weather API.<|assistant|>get_weather`
4. ````python`
5. `tool_call(location='Beijing')`
6. ````<|observation|>`
7. `{"weather": "cloudy", "temperature": 15.6}<|assistant|>`
8. `It's cloudy now in Beijing and the temperature is 15.6 °C.`

# 对话格式：工具调用

- 模型调用工具时给出 `<|observation|>` 作为结束 token
- 例如：
  1. `<|user|>`
  2. `Can you tell me how the weather is in Beijing now?<|assistant|>`
  3. `Sure! I can help with that by querying a weather API.<|assistant|>get_weather`
  4. ````python`
  5. `tool_call(location='Beijing')`
  6. ````<|observation|>`
- 因此，需要检查 `<|user|>` 与 `<|observation|>` 两种终止 token 并作不同处理

# 对话格式：工具调用

1. `<|user|>`
2. `Can you tell me how the weather is in Beijing now?<|assistant|>`
3. `Sure! I can help with that by querying a weather API.<|assistant|>get_weather`
4. ````python`
5. `tool_call(location='Beijing')`
6. ````<|observation|>`

## 如何提取参数：

1. `eval()`，方便但需要注意安全性问题，最好在隔离环境中运行
  - `eval('exec("import os; os.system(\'sudo rm -rf /\')")')`
2. 手动解析 AST
3. 限制能够执行的代码
  1. Transformers Agents: 限制执行的代码只能调用工具和 `print` 函数  
[https://huggingface.co/docs/transformers/transformers\\_agents#code-execution](https://huggingface.co/docs/transformers/transformers_agents#code-execution)



# 对话格式：工具调用

1. `<|user|>`
2. `Can you tell me how the weather is in Beijing now?<|assistant|>`
3. `Sure! I can help with that by querying a weather API.<|assistant|>get_weather`
4. ````python`
5. `tool_call(location='Beijing')`
6. ````<|observation|>`

## 为何使用 Python 代码形式（而非 JSON）？

1. 模型训练的 Python 训练强于 JSON，模型书写正确率高，而 Python 的表达能力是 JSON 的真超集
2. Python 更为灵活，可以接入更多调用形式（目前尚未实现）
  - 考虑模型在调用工具前可进行简单 Python 处理
  - 考虑支持多工具连续调用等功能



# 对话格式：Code Interpreter

固定 `system prompt` 以发挥最好效果：

1. `<|system|>`
2. 你是一位智能AI助手，你叫ChatGLM，你连接着一台电脑，但请注意不能联网。在使用Python解决任务时，你可以运行代码并得到结果，如果运行结果有错误，你需要尽可能对代码进行改进。你可以处理用户上传到电脑上的文件，文件默认存储路径是`/mnt/data/`。

# 对话格式: Code Interpreter

除正常对话外, `<|assistant|>interpreter` 执行代码:

1. `<|user|>`
2. Calculate the 10th Fibonacci number`<|assistant|>`
3. The Fibonacci sequence is defined as each number equaling the sum of the previous two numbers. Therefore, we can write a recursive function to calculate it.`<|assistant|>interpreter`
4. ````python`
5. `def fibonacci(n):`
6.  `if n == 0:`
7.  `return 0`
8.  `elif n == 1:`
9.  `return 1`
10.  `else:`
11.  `return fibonacci(n-1) + fibonacci(n-2)`
12. `fibonacci_10 = fibonacci(9) + fibonacci(8)`
13. `fibonacci_10`
14. ````<|observation|>`
15. `55<|assistant|>`
16. The 10th Fibonacci number is 55.

# 对话格式: Code Interpreter

- 对于图片，直接返回 [Image] 占位符即可：

1. `<|user|>`
2. Draw a heart.`<|assistant|>`
3. `import matplotlib.pyplot as plt`
4. `import numpy as np`
5. `t = np.linspace(0, 2 * np.pi, 1000)`
6. `x = 16 * np.sin(t)**3`
7. `y = 13 * np.cos(t) - 5 * np.cos(2*t) - 2 * np.cos(3*t) - np.cos(4*t)`
8. `plt.figure(figsize=(6,6))`
9. `plt.plot(x, y, 'r')`
10. `plt.title('Heart Shape')`
11. `plt.axis('off')` # Turn off the axis
12. `plt.axis('equal')` # Ensure the shape is not distorted
13. `plt.fill_between(x, y, color='red')` # Fill the heart shape with red color
14. `plt.show()``<|observation|>`
15. [Image]`<|assistant|>`
16. Here's a heart shape drawn using Python and the Matplotlib library. The heart shape is created using parametric equations, and the resulting plot shows the shape of the heart. The heart is filled with a red color, giving it a distinct appearance.

```
plt.figure(figsize=(6,6))
plt.plot(x, y, 'r')
plt.title('Heart Shape')
plt.axis('off') # Turn off the axis
plt.axis('equal') # Ensure the shape is not distorted
plt.fill_between(x, y, color='red') # Fill the heart shape with red color
plt.show()
```

Executing...



Heart Shape



Here's a heart shape drawn using Python and the Matplotlib library. The heart shape is created using parametric equations, and the resulting plot shows the shape of the heart. The heart is filled with a red color, giving it a distinct appearance.



# 对话格式: Code Interpreter

- 与工具调用类似, 执行代码时模型给出 `<|observation|>` 作为终止 token
- 模型可能会在看到 `<|observation|>` 后继续对话和执行代码

# 对话格式: tokenization

- ChatGLM3 的 special token 是防注入的
- 因此, 不能直接通过拼接字符串得到正确的模型输入
- 使用模型的 `chat()`, `stream_chat()` 函数并传入历史、工具、角色等信息进行交互
- 或通过 tokenizer 的 `build_chat_input()` 函数手动构建模型输入

# 对话格式：tokenization

- [https://huggingface.co/THUDM/chatglm3-6b/blob/main/tokenization\\_chatglm.py#L184](https://huggingface.co/THUDM/chatglm3-6b/blob/main/tokenization_chatglm.py#L184)

```
1. def build_chat_input(self, query, history=None, role="user"):
2. if history is None:
3. history = []
4. input_ids = []
5. for item in history:
6. content = item["content"]
7. if item["role"] == "system" and "tools" in item:
8. content = content + "\n" + \
9. json.dumps(item["tools"], indent=4, ensure_ascii=False)
10. input_ids.extend(self.build_single_message(
11. item["role"], item.get("metadata", ""), content
12.))
13. input_ids.extend(self.build_single_message(role, "", query))
14. input_ids.extend([self.get_command("<|assistant|>")])
15. return self.batch_encode_plus([input_ids], return_tensors="pt",
16. is_split_into_words=True)
```

# 对话格式：tokenization

- [https://huggingface.co/THUDM/chatglm3-6b/blob/main/tokenization\\_chatglm.py#L177](https://huggingface.co/THUDM/chatglm3-6b/blob/main/tokenization_chatglm.py#L177)
- 1. `def build_single_message(self, role, metadata, message):`
- 2.  `assert role in ["system", "user", "assistant", "observation"], role`
- 3.  `role_tokens = [self.get_command(f"<|{role}|>")] + self.tokenizer.encode(f"{metadata}\n")`
- 4.  `message_tokens = self.tokenizer.encode(message)`
- 5.  `tokens = role_tokens + message_tokens`
- 6.  `return tokens`
- 注意 `get_command` 函数函数的使用，直接找到角色对应的 special token
- 用户即使输入了形如 `<|user|>` 的内容也不会被编码为 special token



# 对话微调

- 与此前的做法不同, chatglm3-6b 推荐使用**多轮对话格式**进行微调
  - 即将多个不同角色的对话内容**直接拼接**进行 teacher-forcing
  - Special token 的加入使得多轮训练变得容易
  - 不再特殊区别 prompt 和 response
- 需正确配置 loss\_mask, 即哪些 token 的预测需要模型学习
- loss\_mask 的配置依据是推理时的行为
  - 模型自己生成的 token 需要计算 loss
  - 推理系统插入的 token 无需计算 loss

# 对话微调: loss\_mask

- 在对话微调时, 对 loss\_mask 做如下假设
  - `<|system|>` `<|user|>` 和 `<|observation|>`
    - 这三类角色**所预测的 token 不需学习**
    - 注意, 这并不意味着这些 token 不需学习
  - `<|assistant|>`
    - **所预测的 token 可以学习**
    - 也不总需学习

如 few-shot prompt 中的 `<|assistant|>` 角色通常无需学习
- 原因
  - `<|system|>` `<|user|>` 和 `<|observation|>` 总是由推理系统插入
  - `<|assistant|>` **所预测的内容**总是由模型产生



# 对话微调：loss\_mask 示例

`<|system|>` Answer the following questions as best as you can. You have access to the following tools:\n [...]

`<|user|>`\n北京的天气怎么样？

`<|assistant|>`\n我需要调用天气预报工具来获取巴黎的天气信息。

`<|assistant|>`get\_weather\n```\npython\ntool\_call(location="北京")```

`<|observation|>`\n{"temperature\_c": 18, "description": "haze"}

`<|assistant|>`\n根据天气工具的信息，北京的天气是：温度 22 摄氏度，有雾。

`<|user|>`\n这样的天气适合外出活动吗？

`<|assistant|>`\n北京现在有雾，气温较低，建议您考虑一下是否适合外出进行锻炼。

`<|user|>`

高亮部分为需要计算 loss 的 token。

值得注意的是，`<|assistant|>` 后的内容和角色 **token** 都需要计算 loss。

# 微调：代码示例

- 示例数据
  - AdvertiseGen & ToolAlpaca
- 提示
  - 微调后模型的分布发生变化，通用能力和泛化性可能会减弱
  - 加入通用数据（如 ShareGPT）进行混合训练可能缓解通用能力的衰减
  - 若对 Base 模型进行微调，可自行设计生成格式，无需遵循前述格式

# 微调：输入输出 (AdvertiseGen)

**Prompt:** 类型#裤\*版型#宽松\*材质#亚麻\*裤长#九分裤\*裤长#连体裤\*裤款式#勾花镂空\*裤腰型#松紧腰

**Response:** 宽松的连体裤，宽松的版型，上身舒适不紧绷，松紧腰的设计，穿着舒适不勒腰，宽松的版型，穿着舒适不紧绷，九分裤的长度，露出脚踝，更显腿长。镂空的设计，更显时尚。亚麻的面料，穿着舒适透气，上身舒适不闷热。

# 微调：多轮对话 (ToolAlpaca)

Tools

Define your tools in YAML format here:

```
and size.\nParameters: {\n"color\": \"string. One of: [wild, leucistic, albino]. The color of the axolotl (e.g., 'wild', 'leucistic', 'albino', etc.).\", \"gender\": \"string. One of: [male, female]. The gender of the axolotl ('male', 'female').\", \"size\": \"string. One of: [small, medium, large]. The size of the axolotl ('small', 'medium', 'large').\", \"page\": \"integer. The page number for pagination purposes.\"}\n\nOutput: Successful response.\n- Format: application/json\n- Structure: Object{results: Array[Object{url, source, description}], pagination: Object{current_page, total_pages, total_results}}\n\ngetAxolotlFacts: Retrieve interesting facts about axolotls such as their habits, habitats, and physical characteristics.\nParameters: {\n\"category\": \"string. One of: [habits, habitat, physical characteristics]. The category of facts to retrieve (e.g., 'habits', 'habitat', 'physical characteristics').\", \"limit\": \"integer. The maximum number of facts to return.\"}\n\nOutput: Successful response.\n- Format: application/json\n- Structure: Array[Object{fact, source}]\n"
```



Hey, can you show me a random picture of an axolotl?



Calling tool `getRandomAxolotlImage`: Retrieve a random axolotl image with information on the image source. :

```
tool_call()
```

Before

Tools

Define your tools in YAML format here:

```
and size.\nParameters: {\n"color\": \"string. One of: [wild, leucistic, albino]. The color of the axolotl (e.g., 'wild', 'leucistic', 'albino', etc.).\", \"gender\": \"string. One of: [male, female]. The gender of the axolotl ('male', 'female').\", \"size\": \"string. One of: [small, medium, large]. The size of the axolotl ('small', 'medium', 'large').\", \"page\": \"integer. The page number for pagination purposes.\"}\n\nOutput: Successful response.\n- Format: application/json\n- Structure: Object{results: Array[Object{url, source, description}], pagination: Object{current_page, total_pages, total_results}}\n\ngetAxolotlFacts: Retrieve interesting facts about axolotls such as their habits, habitats, and physical characteristics.\nParameters: {\n\"category\": \"string. One of: [habits, habitat, physical characteristics]. The category of facts to retrieve (e.g., 'habits', 'habitat', 'physical characteristics').\", \"limit\": \"integer. The maximum number of facts to return.\"}\n\nOutput: Successful response.\n- Format: application/json\n- Structure: Array[Object{fact, source}]\n"
```



Hey, can you show me a random picture of an axolotl?



I need to find a random axolotl image.



Calling tool `getRandomAxolotlImage` :

```
tool_call()
```

After