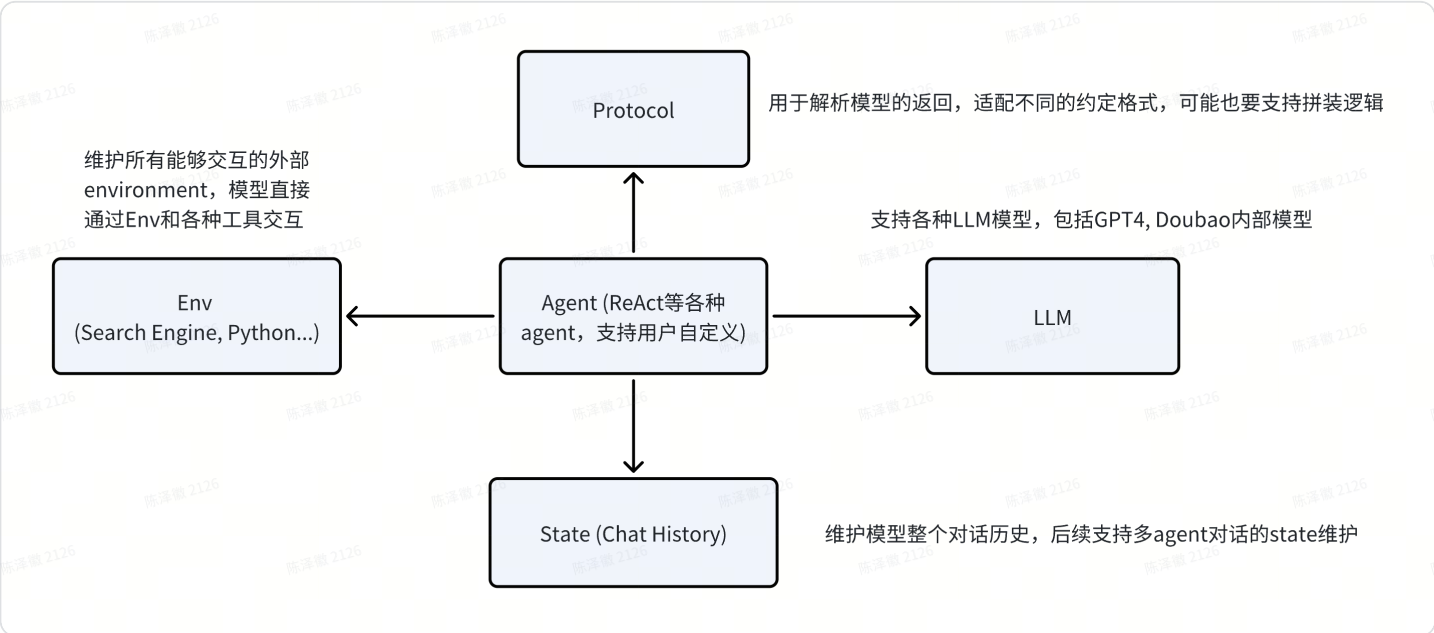
Groot框架设计

https://code.byted.org/seed/groot_new

Groot提供了一套简单易用的agent框架，保留了原版groot的大部分设计思路，和原版groot不同的地方在于它做了更多的解耦合。

主要更新点如下:

1. 把client和env合并了，目前的client除了llm，对于agent来说都是实际需要执行的agent，都放到action/下
2. 通过Env来管理所有工具的description和调用，统一做mapping，而不是在agent里面写处理逻辑，同时支持了docstring直接从注释里面提取工具description，不用单独写
3. llm单独抽象出来，构造多智能体时允许不同LLM作为不同的agent backend，而不是与Prompt耦合在一起
4. 把llm的response提取action的这一步抽象出一个protocol来做，支持用户根据prompt自行定义action抽取逻辑，而不是耦合在State里面做



设计介绍

action类

Env

用于管理各种action，同时从action中抽取docstring自动生成API description

主要两个接口 get_actions_info用于生成system prompt，另外__call__用于统一管理执行action

```
 def get_actions_info(self) -> List[Dict]:
    """
    Retrieves the information of all available actions.

    Returns:
        List[Dict]: A list of dictionaries containing descriptions of each action.
                    For toolkit actions, API-specific descriptions are included.
    """

 def __call__(self, name: str, parameters: dict) -> Any:
    """
    Executes a specific action or API method by its name.

    Args:
        name (str): The name of the action or the API method in the format 'action.api_method'.
                    If only the action name is provided, 'run' is the default method.
        parameters (dict): A dictionary of parameters to pass to the action or API method.

    Returns:
        Any: The result of the action execution, or an error message if the action does not exist.
    """
```

其他各种Action

两种情况

1. 只有一个action，这种时候只要实现run的逻辑就可以
2. 这个action支持多个API，这种时候实现对应api的逻辑，不用实现run的逻辑

注意点:

1. 对于要作为API的function加上@tool_api的装饰器，如果不需要指定参数类型等信息，则需要加上参数 `explode_param=False` , 参考code executor这个action
2. 因为所有的API description直接从注释中抽取(抽取逻辑: `action/base_action.py`)，请在实现对应api方法时把注释写清楚，如下

```python
@tool_api
def select(self, select_ids):
    """在使用WebSearch.search搜索到相关内容后，可以给定对应ID获得详细的搜索内容.

    Args:
        select_ids (List[int]): list of index you want to open, select at least 3, at most 5 items to open, each item must be integer.
    """
    for idx in select_ids:
        if idx not in self.search_results:
            return f"{idx}不存在在搜索结果中，请确保输入的id在搜索结果中"
    for idx in select_ids:
        html, _  = self.open(self.search_results[idx]['url'])
        self.search_results[idx]['content'] = html
    return self.search_results
```

llm类

支持generate接口和chat接口，generate的输入是str，不做拼接，chat支持在内部做拼接（目前只支持doubao），即传入是

同时gpt也看成是一种llm，这样后续方便支持huggingface等其他各种接口的llm

state类

用户维护llm对话历史内容，和之前的设计保持一致

protocol类

用户处理llm返回时的各种格式逻辑，提取对应的格式化信息

```python
import json

class BaseProtocol:
    def __init__(self,
                 special_tokens=dict(
                    action_start_token='',
                    action_end_token=''),
                 ):
        self.special_tokens = special_tokens

    def parse(self, text):
        if self.special_tokens['action_start_token'] in text:
            # split CoT & action
            text, action_str = text.split(self.special_tokens['action_start_token'])
```

```
            action_str = action_str.split(self.special_tokens['action_end_token'])[0]
            if action_str.startswith('```'):    # NOTE specially handle for PythonInterpreter
                action = dict(name='PythonInterpreter', parameters=dict(code=action_str))
            else:
                action = json.loads(action_str)
        else:
            action = None
        return text, action

class DoubaoProtocol(BaseProtocol):
    def __init__(self,
                 special_tokens=dict(
                    action_start_token='<|FunctionCallBegin|>',
                    action_end_token='<|FunctionCallEnd|>'
                 )):
        super().__init__(special_tokens)
```

agent类

这里用一个react的执行逻辑作为example，agent本身不做很多要求，用户用上述基类直接实现agent即可，提供尽可能大的灵活多, 输入就是user_prompt

```
def run(self, inputs: Optional[str] = None) -> str:
    """
    Run the ReAct agent in an interactive loop.

    Args:
        inputs (Optional[str]): The user input to start the conversation.

    Returns:
        str: Final response after completing the interaction.
    """
    # Add user input to conversation state
    self.state.add(role='user', content=inputs)

    for turn in range(self.max_turn):
        # Get response from the language model
        llm_response = self.llm.chat(self.state.history)
        print(colored(f"Assistant: {llm_response}", 'blue'))

        # Add LLM response to conversation state
        self.state.add(role='assistant', content=llm_response)
```

```
21
22          # Parse the LLM response to determine if an action is required
23          message, action = self.protocol.parse(llm_response)
24
25          # If no action is required, assume the conversation has reached the
    terminal state
26          if action is None:
27              return message
28
29          # Execute action in the environment and capture the response
30          try:
31              env_response = self.env(**action)
32          except Exception as e:
33              env_response = f"Error during environment interaction: {str(e)}"
34          print(colored(f"Environment: {env_response}", 'green'))
35
36          # Add environment response to conversation state
37          self.state.add(role='tool', content=env_response)
38
39      # If max turns reached without termination, return this message
40      return "Not finished"  # Possibly force direct summary if no terminal
    state is reached
41
```

使用示例

```
1  from groot.agent import ReAct
2  from groot.action import WebSearch, CodeExecutor, Env
3  from groot.llm import DoubaoAPI
4  from groot.agent.protocol import DoubaoProtocol
5
6  model_name = 'Seed-32B-RL-P6.0.0_D7.3.0_T17119B-SFT29.0.0_RM7.19.0_RL1.0.0-
   C3.0.1-rl27.turn1.v1.tp8'
7  env = Env([WebSearch(), CodeExecutor()])
8  llm = DoubaoAPI(
9      mode='psm',
10     psm_cfg=dict(
11         model_name=model_name,
12         psm='data.seed.rl_human_eval',
13         idc='lq',
14         llm_cluster=model_name,
15     )
16 )
17 protocol = DoubaoProtocol()
```

```
agent = ReAct(llm=llm, env=env, protocol=protocol)
agent.run("阿里巴巴在 2023 年的电商业务交易额是多少？同时对比一下京东在同年的电商业务交易额，然后评估一下两家企业在电商领域的核心竞争力分别是什么")
```