Custom chain

To implement your own custom chain you can subclass `Chain` and implement the following methods:

```
from __future__ import annotations

from typing import Any, Dict, List, Optional

from pydantic import Extra

from langchain.schemea import BaseLanguageModel
from langchain.callbacks.manager import (
    AsyncCallbackManagerForChainRun,
    CallbackManagerForChainRun,
)
from langchain.chains.base import Chain
from langchain.prompts.base import BasePromptTemplate


class MyCustomChain(Chain):
    """
    An example of a custom chain.
    """

    prompt: BasePromptTemplate
    """Prompt object to use."""
    llm: BaseLanguageModel
    output_key: str = "text"  #: :meta private:
```

```
    class Config:
        """Configuration for this pydantic object."""

        extra = Extra.forbid
        arbitrary_types_allowed = True

    @property
    def input_keys(self) -> List[str]:
        """Will be whatever keys the prompt expects.

        :meta private:
        """
        return self.prompt.input_variables

    @property
    def output_keys(self) -> List[str]:
        """Will always return text key.

        :meta private:
        """
        return [self.output_key]

    def _call(
        self,
        inputs: Dict[str, Any],
        run_manager: Optional[CallbackManagerForChainRun] = None,
    ) -> Dict[str, str]:
        # Your custom chain logic goes here
        # This is just an example that mimics LLMChain
        prompt_value = self.prompt.format_prompt(**inputs)

        # Whenever you call a language model, or another chain, you should pass
        # a callback manager to it. This allows the inner run to be tracked by
        # any callbacks that are registered on the outer run.
        # You can always obtain a callback manager for this by calling
        # `run_manager.get_child()` as shown below.
        response = self.llm.generate_prompt(
            [prompt_value], callbacks=run_manager.get_child() if run_manager else None
        )

        # If you want to log something about this run, you can do so by calling
        # methods on the `run_manager`, as shown below. This will trigger any
        # callbacks that are registered for that event.
        if run_manager:
            run_manager.on_text("Log something about this run")

        return {self.output_key: response.generations[0][0].text}

    async def _acall(
        self,
        inputs: Dict[str, Any],
        run_manager: Optional[AsyncCallbackManagerForChainRun] = None,
    ) -> Dict[str, str]:
        # Your custom chain logic goes here
        # This is just an example that mimics LLMChain
        prompt_value = self.prompt.format_prompt(**inputs)

        # Whenever you call a language model, or another chain, you should pass
        # a callback manager to it. This allows the inner run to be tracked by
        # any callbacks that are registered on the outer run.
        # You can always obtain a callback manager for this by calling
        # `run_manager.get_child()` as shown below.
        response = await self.llm.agenerate_prompt(
            [prompt_value], callbacks=run_manager.get_child() if run_manager else None
        )
        # If you want to log something about this run, you can do so by calling
        # methods on the `run_manager`, as shown below. This will trigger any
        # callbacks that are registered for that event.
        if run_manager:
            await run_manager.on_text("Log something about this run")

        return {self.output_key: response.generations[0][0].text}

    @property
    def _chain_type(self) -> str:
        return "my_custom_chain"
```

```
from langchain.callbacks.stdout import StdOutCallbackHandler
from langchain.chat_models.openai import ChatOpenAI
from langchain.prompts.prompt import PromptTemplate


chain = MyCustomChain(
    prompt=PromptTemplate.from_template("tell us a joke about {topic}"),
    llm=ChatOpenAI(),
)

chain.run({"topic": "callbacks"}, callbacks=[StdOutCallbackHandler()])
```

```


    > Entering new MyCustomChain chain...
    Log something about this run
    > Finished chain.
```

```
'Why did the callback function feel lonely? Because it was always waiting for someone to call it back!'
```