LangChain – техническо описание

Версия 2 / 30 май 2026 г.

Съдържание

Въведение

Настоящото техническо описание се фокусира върху използването на вградени изпълними компоненти (runnables), имплементацията на нови такива и тяхното свързване във изпълними поредици (runnable sequences) или вериги (chains, откъдето идва и името на програмната рамка LangChain). Даденият примерен код е на скриптовия език Python.

Инсталация

LangChain библиотеката е разделена на няколко пакета, съдържащи различни компоненти:

- `langchain` и `langchain-core` – основни класове, интерфейси и функции за изпълними компоненти, модели, агенти, енбединги, зареждане и запазване на документи, шаблони за заявки и др.
- `langchain-text-splitters` – класове за разделяне на текст написан различни формати (HTML, JSON, LaTeX, Python и др.)
- `langchain-community` – разнообразие от класове и функции, създадени от други разработчици
- `langchain-experimental` – експериментални класове и функции
- пакети за интеграция с външни доставчици на различни функционалности чрез съответния приложно-програмен интерфейс (Application Programming Interface - API), напр. OpenAI, Ollama, AWS, Chroma и др.

За инсталация на LangChain в общия случай е необходимо инсталирането на пакетите `langchain`, `langchain-core`, `langchain-community` и желаните пакети за интеграция. Имената на някои от пакетите за интеграция са описани по-долу.

```
pip install langchain langchain-core langchain-community
```

Поредици от изпълними компоненти

Подобно на много командни интерпретатори (Bash, Zsh, PowerShell и др.), LangChain използва оператора `|` (pipe, пайп, тръба) за свързване на компоненти във верига, през която да бъде обработена информация в отделни стъпки. В тази верига резултатът от един компонент се прехвърля като вход за следващи.

Изпълними компоненти

В контекста на LangChain, изпълними компоненти (runnables) са всички инстанции на класове, които имплементират интерфейса `Runnable`. В този интерфейс са декларирани методите, позволяващи създаването на вериги:

- `invoke`, `ainvoke` – трансформиране на прост вход в изход, т.е. изпълнение на основната функция на компонента.
- `batch`, `abatch` – ефикасно трансформиране на множество входове чрез паралелно изпълнение на `invoke` или `ainvoke` съответно.
- `stream`, `astream` – предаване на парчета от резултата в момента, в който са генерирани.
- `astream_log` – предаване всичкия изход от изпълнението на функцията на компонента в реално време.

Две допълнителни функционалности на класа `RunnableSerializable` позволяват оптимизация на изпълнението на заявките:

- Групиране (`batch`) – по подразбиране `batch` изпълнява метода `invoke` паралелно, като използва пул от нишки.
- Асинхронно изпълнение (`async`) – всички методи, започващи с „a-“ са асинхронни, като те изпълняват синхронния вариант на съответния метод чрез пула от нишки от библиотеката `asyncio`.

Методите за оптимизация чрез групиране и асинхронно изпълнение могат да бъдат предефинирани, за да използват други стратегии и библиотеки.

Сред различните видове изпълними компоненти в това описание се разглеждат:

- Компоненти за използване на модели от различни доставчици, като Ollama, OpenAI, Hugging Face
- Шаблони за заявки
- Компоненти за обработка на резултат (парсъри)
- Хранилища (бази данни) за вграждания
- Персонализирани и нови компоненти

Изпълними поредици

Споменатите по-горе компоненти могат да бъдат свързвани в последователност, като изхода на един компонент се използва за вход на следващия, създавайки изпълнима поредица. Това се прави чрез метода `pipe` на обект от тип `RunnableSerializable` или оператора `|`, който е предефиниран за създаване на изпълнима поредица чрез извикването на този метод. Резултатът на изпълнението на метода или оператора е инстанция на класа `RunnableSequence`.

```
from langchain_core.prompts import PromptTemplate
from langchain_ollama import OllamaLLM

prompt = PromptTemplate.from_template("Преведи '{phrase}' на български")
model = OllamaLLM(model="llama3")
chain = prompt | model  # Еквивалентно на prompt.pipe(model)

result = chain.invoke("learning curve")
print(result)
```

```
Крива на учене
```

Класът `RunanbleSequence` сам по себе си е наследник на абстрактния клас `RunnableSerializable`, което означава, че изпълнимите поредици могат да бъдат разглеждани и използвани като сложни изпълними компоненти, притежаващи описаните по-горе методи.

Модели

Базови класове в LangChain

LangChain поддържа различни видове модели според тяхното използване:

- Модели за генериране на текст с входни данни текст, наследяващи базовия клас `BaseLLM`

- Модели за генериране на текст с входни данни поредица от съобщения (чат история), наследяващи базовия клас `BaseChatModel`
- Модели за генериране на вграждания с входни данни текст, които наследяват базовия клас `BaseModel` и имплементират интерфейса `Embeddings`

Тези класове и интерфейси са наследени и имплементирани в различни пакети за интеграция с доставчици на големи езикови модели. Подробен списък с поддържаните интеграции за модели е наличен на адрес https://python.langchain.com/docs/integrations/providers/.

Ollama

Пакетът за интеграция предоставя класове за ползване на Ollama сървъра, локален или на друг адрес, за изпълнение на заявки към модели както от библиотеката на Ollama, така и модели от други източници, включително и локални за Ollama сървъра, записани във формати GGUF и Safetensors. За работа с тези класове са необходими библиотеката за интеграция `langchain-ollama` и библиотеката `ollama`.

```
pip install langchain-ollama ollama
```

В пакета `langchain_ollama` са налични три класа за работа с Ollama – `OllamaLLM` и `ChatOllama` (за генериране на текст), и `OllamaEmbeddings` (за генериране на вграждания – embeddings). Тези класове имплементират връзка към Ollama сървъра, който работи локално на адрес `http://127.0.0.1:11434` (`localhost` порт 11434), или който е зададен в променливата на средата `OLLAMA_HOST`.

Модели за генериране на текст

Интеграцията за Ollama предоставя два класа за използване на модели за генериране на текст:

- `OllamaLLM` – генериране на текст с входни данни текст
- `ChatOllama` – генериране на текст с входни данни поредица от съобщения

Обекти от класовете `OllamaLLM` и `ChatOllama` се създават чрез конструкторите на класовете с параметри:

- `model` – името и тагът на модела, разделени с `:` (ако моделът е от библиотеката на Ollama), който да бъде зареден, напр. `llama3.2:1b`. Ако тагът е пропуснат, се приема, че е `latest` (напр. `llama3.2` е еквивалентно на `llama3.2:latest`).
- `base_url` – пътят или адресът, където се намира модела, в случай, че не е от библиотеката на Ollama.

Възможно задаването на допълнителни параметри за работата на избрания модел, специфични за рамката на Ollama:

- `temperature` – стойност между 0 и 1, която определя "креативността" на модела. По-ниска стойност води до по-консистентни отговори.
- `num_predict` – ако е зададен, определя максималния брой на генерираните токъни, като в противен случай:
  - стойност по подразбиране – 128 токъна
  - стойност -1 – неограничено генериране
  - стойност -2 – генериране на токъни до запълване на контекста

- и други – списък с възможни параметри може да бъде намерен в GitHub проекта на Ollama на адрес https://github.com/ollama/ollama/blob/main/docs/modelfile.mdx/#valid-parameters-and-values

```
from langchain_ollama import OllamaLLM, ChatOllama

textLLM = OllamaLLM(
    model="llama3.2:3b",
    temperature=0.5,
    num_predict=250
)

chatLLM = ChatOllama(
    model="llama3.2:3b",
    temperature=0.5,
    num_predict=250
)
```

Класът `OllamaLLM` представлява базата за използване на модели. Той позволява подаването на текст със заявка (промпт) и генерирането на текст като отговор. Класът `ChatOllama` позволява подаване на текст със заявка (промпт) или задаване на контекст – история на разговор между потребителя и модела – като списък с двойки с формат `(<роля>, <съобщение>)`, където роля може да бъде `system`, `human` или `ai`, или обекти от тип `SystemMessage`, `HumanMessage` и `AIMessage`, а резултатът от изпълнението на модела е следващото съобщение в чата.

```
textResult = textLLM.invoke("You are a question answering assistant. "
    "Your topic is \"Artificial Intelligence\". "
    "If the question is out of topic, explain this without answering. "
    "Finish within three sentences."
    "Question: What is a neural network?")  # str

chatResult = chatLLM.invoke([
    {
        "role": "system",
        "content": "You are a question answering assistant. "
                   "Your topic is \"Artificial Intelligence\". "
                   "If the question is out of topic, explain this without
answering. "
                   "Finish within three sentences."},
    {
        "role": "human",
        "content": "What is a neural network?"
    }
])  # BaseMessage

print("Text:", textResult)
print("Chat:", chatResult.text)
```

```
Text: A neural network is a fundamental component of Artificial Intelligence (AI)
and Machine Learning (ML). It's a computational model inspired by the human
brain's structure and function, composed of interconnected nodes or "neurons" that
process and transmit information. In the context of AI, neural networks are used
to recognize patterns in data, make predictions, and learn from experience.
Chat: A neural network is a machine learning model inspired by the structure and
function of the human brain. It consists of layers of interconnected nodes or
"neurons" that process and transmit information, allowing the network to learn and
make decisions based on data patterns and relationships. Neural networks are
```

```
widely used in various applications, including image recognition, natural language
processing, and speech recognition.
```

Модели за генериране на вграждания

В библиотеката на Ollama са достъпни и някои модели за генериране на вграждания, които се използват в приложения като генериране, подпомогнато от извличане (retrieval-augmented generation, RAG). Класът `OllamaEmbeddings` се инициализира със следните параметри:

- `model` – името и тагът на модела (ако моделът е от библиотеката на Ollama), който да бъде зареден, напр. `nomic-embed-text:v1.5`. Ако тагът е пропуснат, се приема, че е `latest`.
- `base_url` – пътят или адресът, където се намира моделът, в случай, че не е от библиотеката на Ollama.

Както при текстовите модели, за моделите за вграждания също е възможно задаването на специфични за рамката на Ollama допълнителни параметри.

Hugging Face

Интеграцията на Hugging Face предлага класове за използване на модели за генериране на текст и такива за генериране на вграждания. Във всяка от тези групи се открояват и два метода за използване на моделите спрямо локацията на тяхната работа – в Hugging Face Hub или локално. Основният пакет за интеграция с Hugging Face е `langchain-huggingface`. За работа с Hugging Face Hub е допълнително необходим пакет `huggingface_hub`, за локално изпълнение на заявки за генериране на текст – `transformers`, а за генериране на вграждания – `sentence_transformers`.

```
pip install langchain-huggingface huggingface_hub transformers \
sentence_transformers
```

Модели за генериране на текст

Интеграцията за Hugging Face предлага два класа за работа с модели за генериране на текст: `HuggingFaceEndpoint` и `HuggingFacePipeline`.

Класът `HuggingFaceEndpoint` позволява използване на модели, които работят с ресурсите в Hugging Face Hub и изисква допълнителния пакет `huggingface_hub`. Създаването на инициализиран обект от класа се извършва чрез конструктора на класа. Основни параметри за инициализация на класа са:

- `repo_id` – идентификацията на хранилището (repository, репозитори) на използвания модел в HuggingFace. Задължителен параметър ако не са зададени `endpoint_url` и променливата на средата `HF_INFERENCE_ENDPOINT`.
- `endpoint_url` – адрес, на който да бъдат намерени файловете на използвания модел. Задължителен параметър ако не е зададен `repo_id` и променливата на средата `HF_INFERENCE_ENDPOINT`.
- `huggingfacehub_api_token` – токън за достъп до API на HuggingFace, ако не е зададен в променливата на средата `HUGGINGFACEHUB_API_TOKEN`
- `max_new_tokens` – максималният брой на генерираните токъни.

- `temperature` – стойност между 0 и 1, която определя "креативността" на модела. Стойност 0 води до по-консистентни отговори.

Класът `HuggingFacePipeline` позволява използването на модели от библиотеката на Hugging Face локално чрез канали (pipelines). Наличните канали са `text-generation`, `text2text-generation`, `image-text-to-text-generation`, `summarization` и `translation`. Създаването на инициализиран обект от класа се извършва чрез статичния метод `from_model_id` или с предварително създаден канал чрез функцията `pipeline` от библиотеката `transformers`.

Основни параметри за създаването на нов обект от класа със статичния метод `from_mode_id` са:

- `model_id` - името на използвания модел. Ако не е зададен, се използва моделът по подразбиране за избрания канал. Ако такъв няма, се използва моделът определен от променливата на средата `DEFAULT_MODEL_ID`.
- `task` - избраният канал.
- `model_kwargs` - параметри на модела.
- `pipeline_kwargs` - параметри на канала.

Клас за поддръжка на чат контекст

В пакетът за интеграция на Hugging Face е дефиниран класът `ChatHuggingFace`. За неговото използване е необходим вече деклариран и инициализиран обект от един от класовете `HuggingFaceEndpoint` и `HuggingFacePipeline`. Обект от класа `ChatHuggingFace` се създава чрез конструктора на класа, който има един основен параметър – `llm`, чрез който се подава предварително дефинираният обект за модел. С допълнителен параметър `verbose` от булев тип се определя дали да бъде записан резултатът на стандартния изход при извикване.

Модели за генериране на вграждания

Класът за използване на модели за генериране на вграждания от Hugging Face е `HuggingFaceEmbeddings` и за работа с него е необходим пакетът `sentece_transformers`. Обект на класа се създава чрез конструктор, който приема следните параметри:

- `model_name` – името на модела
- `model_kwargs` – параметри на модела
- `encode_kwargs` – параметри при извикеане на метода `encode` на модела, сред които:
    - `prompt` – заявката, която да бъде използвана при кодирането. Заявката се прибавя преди изречението и вграждането се прави за резултата
    - `output_value` – определя какъв да бъде резултатът на кодирането, като стойността се подава като низ: `"sentence_embeddings"` – вграждания за изречения; `"token_embeddings"` – вграждания за думи и части от думи (токъни); `None` – за всички резултати

OpenAI

Моделите на компанията OpenAI са сред най-известните и най-често използвани големи езикови модели със затворен код чрез техния приложно-програмен интерфейс.

Интеграцията на LangChain за OpenAI моделите включва класове за генериране на текст и вграждания.

```
pip install langchain-openai
```

За да може LangChain да работи с интерфейса на OpenAI, е необходимо запазването на ключа (токъна) за достъп в променливата на средата `OPENAI_API_KEY` в случай, че не е подаден като инициализиращ параметър в конструктор.

Модели за генериране на текст

Пакетът за интеграция на LangChain с OpenAI предлага четири класа за работа с модели за генериране на текст, разделени в зависимост от поддръжката на чат контекст и мястото на разположение на модела – на системите на OpenAI или като инсталация в Microsoft Azure – `OpenAI`, `ChatOpenAI`, `AzureOpenAI` и `AzureChatOpenAI`. Информация за инсталация на OpenAI система в Microsoft Azure е налична на https://learn.microsoft.com/en-us/azure/ai-services/openai/how-to/create-resource.

Обекти на класовете на OpenAI се създават чрез конструктор. Основни параметри за инициализация на класовете са:

- `model` – името на модела, който да бъде използван.
- `temperature` – стойност между 0 и 1, която определя "креативността" на модела. Стойност 0 води до по-консистентни отговори.
- `max_tokens` – максималният брой токъни, които моделът може да генерира.
- `logprobs` – определя дали моделът да върне стойностите на `logprobs`.
- `stream_options` – определяне на параметри за стрийминг.
- `timeout` – определя максимално допустимото време, което генерирането на текст може да отнеме.
- `max_retries` – максималният брой опити за генериране на текст, след което генерацията се счита за неуспешна.
- `api_key` – ключ за използване на интерфейса на OpenAI. Ако не е подадено, се използва стойността на променливата на средата `OPENAI_API_KEY` или `AZURE_OPENAI_API_KEY` за класовете за връзка с OpenAI инсталация в Azure.
- `base_url` – базов URL адрес на интерфейса за заявки. Подава се само ако се използва прокси или емулация.
- `organization` – OpenAI идентификатор на организация. Ако не е подадено, се използва стойността на променливата на средата `OPENAI_ORG_ID`.

Допълнителни параметри за класовете `ChatOpenAI` и `AzureChatOpenAI`, поддържащи чат контекст:

- `use_responses_api` – определя дали да бъде използван по-новият програмен интерфейс за услуги Responses API вместо интерфейсът по подразбиране Completions API.

Допълнителни параметри за класовете `AzureOpenAI` и `AzureChatOpenAI`, поддържащи използването на модели от Azure инсталации:

- `azure_endpoint` – точка за достъп на OpenAI интерфейса в инсталацията на Azure. Ако не е подадено, се използва стойността на променливата на средата `AZURE_OPENAI_ENDPOINT`.

- `openai_api_version` – версия на OpenAI интерфейса в инсталацията на Azure. Ако не е подадено, се използва стойността на променливата на средата `OPENAI_API_VERSION`. Стойността по подразбиране е `'2023-05-15'`.

Параметри, които не са директно поддържани от конструкторите, но са подадени при инициализацията на обекта за модел, напр. `model = OpenAI(<параметри на конструктура>..., <доп. параметър> = <сотйност>)`, се подават като част от заявката към интерфейса на OpenAI при всяко изпълнение на метода `invoke(<вход>...)`. Алтернативно, тези параметри могат да бъдат подадени за всяка отделна заявка при извикване на метода като `model.invoke(<вход>, <доп. параметър> = <стойност>)`.

Модели за генериране на вграждания

Пакетът за интеграция на LangChain с OpenAI съдържа два класа за работа с модели за генериране на вграждания – `OpenAIEmbeddings` и `AzureOpenAIEmbeddings`, вторият от които е специфичен за използване на интерфейса на OpenAI в инсталация в Azure. Инициализирането на обекти от двата класа се извършва чрез техните конструктори. Основни параметри за инициализация са:

- `model` – името на модела, който да бъде използван.
- `dimensions` – брой на измеренията за генерираните вграждания. Поддържа се от модели `'text-embedding-3'` и по-нови.
- `api_key` – ключ за използване на интерфейса на OpenAI. Ако не е подадено, се използва стойността на променливата на средата `OPENAI_API_KEY` или `AZURE_OPENAI_API_KEY` за класовете за връзка с OpenAI инсталация в Azure.
- `organization` – OpenAI идентификатор на организация. Ако не е подадено, се използва стойността на променливата на средата `OPENAI_ORG_ID`.
- `max_retries` – максималният брой опити за генериране на текст, след което генерацията се счита за неуспешна.
- `request_timeout` – определя максимално допустимото време, което заявката за генериране на вграждания може да отнеме.

Шаблони за заявки

LangChain библиотеката `langchain-core` включва няколко класа, чрез които могат да бъдат дефинирани шаблони за създаване на заявки към модели за генериране на текст. Класовете за шаблони за заявки `PromptTemplate` и `ChatPromptTemplate` съответстват на видовете модели – за генериране с вход текст и с вход поредица от съобщения (чат).

Създаването на обект на класа `PromptTemplate` може да бъде извършено чрез статичните методи `from_template`, `from_examples` и `from_file` или чрез конструктора на класа.

- `from_template` – създава шаблон от текстов низ (зададен с параметър `template`) с един от три синтаксиса (зададен в параметър `template_format`):
    - `f-string` – базов вариант на Python f-string, синтаксис по подразбиране. Синтаксисът позволява създаване на шаблон чрез използване на динамични части с единични фигурни скоби (`{…}`), които да бъдат заменени с подадени стойности при генериране на заявка. Повече информация за използването на формата в LangChain е налична на адрес https://docs.langchain.com/langsmith/prompt-template-format#f-string-syntax

    - `mustache` – статичен език за шаблони, използващ нотация с двойни фигурни скоби (`{{…}}`) и позволяващ сложни структури от данни, като речници и масиви. Езикът позволява използване на коментари, които не се включват в генерираната заявка, и структури, наречени секции (sections), които могат да бъдат използвани за цикли (при данни масив), или подобно на условен оператор (при други данни). Повече информация за използването на формата в LangChain е налична на адрес https://docs.langchain.com/langsmith/prompt-template-format#mustache-syntax
    - `jinja2` – динамичен език за шаблони, подобен на mustache, който позволява по-високо ниво на логическо програмиране със структури за цикли, условни оператори, изчисления и др. Използването на формата не е препоръчано, поради възможността за изпълнение на неприятелски код.
- `from_examples` – създава шаблон от списък с текстови примери в параметър `examples`. Общата рамка на шаблона се определя от префикс (`prefix`), суфикс (`suffix`) и разделител за примерите в заявката (`example_separator`). Параметърът `input_variables` задава списък с имената на параметрите, които шаблонът ще очаква. Форматът на шаблона е f-string.
- `from_file` – зарежда шаблон от файл, чиято локация е зададена в параметър `template_file`. Кодирането на файла е определено в параметър `encoding`. Очакваният формат на шаблона е f-string.
- конструктор – подобно на метод `from_template`, като шаблонът се задава е параметър `template`.

След създаване на обекта на класа `PromptTemplate`, той може да бъде използван са създаване на заявка чрез метода `format_prompt`, на който се подават именувани параметри според зададения шаблон. Методът връща текстов низ, създаден на базата на шаблона, неговия формат (f-string, mustache или jinja2) и подадените като параметри данни. По подобен начин, чрез метода `partial` и стойности за някои от параметрите на шаблона, може да бъде направен нов шаблон от вече съществуващ с частично попълнени параметри.

Класът `ChatPromptTemplate` се използва за дефиниране на шаблон на заявка, представляваща поредица от съобщения (чат) – двойки от роля и текст. Ролите могат да бъдат:

- `system` – система (за задаване на начални инструкции към модела)
- `human` – човек (потребител)
- `ai` – модел (чатбот), представляващи предишни отговори на модела; с тази роля ще бъде и отговора на модела
- `placeholder` – специална роля само за шаблони, която позволява задаването на чат като стойност на параметър за генериране на заявка от шаблона; съдържанието на съобщението за тази роля трябва да е единствено името на параметъра, напр. `"{content}"`

Подобно на `PromptTemplate`, текстът на съобщенията в шаблона може да бъде дефиниран като динамичен в трите поддържани формата – `f-string`, `mustache` и `jinja2`. Класът се инициализира чрез някой от следните методи:

- `from_template` – с параметър низ от символи, като се приема, че това образува чат със зададения низ като съобщение и роля `human`
- `from_messages` – с параметър чат история
- конструктор – подобно ма метода `from_messages`, като шаблонът се задава в параметъра `messages`

След създаване на обекта на класа `ChatPromptTemplate`, той може да бъде използван са създаване на заявка чрез метода `format_prompt`, на който се подават именувани параметри според зададения шаблон. Методът връща списък от съобщения с определените роли, създаден на базата на шаблона, неговия формат (f-string, mustache или jinja2) и подадените като параметри данни. Чрез метода `partial` и стойности за някои от параметрите на шаблона, може да бъде направен и нов шаблон от вече съществуващ с частично попълнени параметри.