

검색기(Retriever)



1. 검색기(Retriever) 개념

LangChain의 **검색기(Retriever)** 는 RAG(Retrieval-Augmented Generation) 구조에서 **질문과 관련된 문서나 정보 조각을 찾아주는 핵심 모듈**이다.

즉, "지식 검색 엔진" 역할을 하며, LLM이 답변을 생성하기 전에 참고할 문맥(context)을 찾아준다.

1. Retriever의 기본 개념

Retriever는 단순히 **문서를 검색**하는 것이 아니라, ****질문과 의미적으로 유사한 문서 조각(청크)****을 찾아 LLM에게 전달한다.

이 과정은 다음과 같은 단계로 구성된다:

1. 문서 임베딩(embedding)

문서를 벡터(숫자 형태)로 변환해 데이터베이스에 저장한다.

2. 질문 임베딩

사용자의 질문도 같은 방식으로 벡터로 변환한다.

3. 유사도 검색(similarity search)

질문 벡터와 문서 벡터 사이의 거리(코사인 거리 등)를 계산해 가장 유사한 문서를 찾는다.

4. 검색 결과 반환

상위 N개의 문서를 Retriever가 반환 → LLM이 이를 참고해 답변을 생성.

2. 주요 Retriever 종류

Retriever 종류

설명

VectorStoreRetriever

가장 일반적인 형태. FAISS, Chroma, PostgreSQL(pgvector) 등의 벡터 DB에서 검색 수행.

MultiQueryRetriever

LLM을 사용해 질문을 여러 형태로 변환 후, 다양한 질의어로 검색을 수행해 더 풍부한 결과 획득.

ParentDocumentRetriever

문서를 “청크 + 부모문서” 구조로 관리하여, 작은 청크에서 검색해도 원본 전체 문서 문맥을 함께 반환.

ContextualCompressionRetriever

검색된 결과를 다시 요약·압축하여, 불필요한 부분을 제거한 후 LLM에 전달.

SelfQueryRetriever

LLM이 직접 질의어를 SQL 또는 벡터 검색 쿼리 형태로 생성하여 고급 필터링 수행.

TimeWeightedVectorStoreRetriever

“가장 최근 문서일수록 우선”하도록 가중치를 주는 시계열 기반 검색기.

3. 기본 예제: VectorStoreRetriever

python

```
from langchain_community.vectorstores import FAISS
from langchain_openai import OpenAIEmbeddings
from langchain_core.documents import Document

# 1. 문서 임베딩
docs = [
    Document(page_content="LangChain은 LLM 애플리케이션 개발 프레임워크입니다."),
    Document(page_content="Retriever는 문서에서 관련 정보를 검색합니다."),
]
embeddings = OpenAIEmbeddings(model="text-embedding-3-small")

# 2. 벡터스토어 생성
db = FAISS.from_documents(docs, embeddings)

# 3. 검색기 생성
retriever = db.as_retriever(search_kwargs={"k": 2})

# 4. 질의
query = "LLM에서 Retriever의 역할은?"
results = retriever.invoke(query)

for doc in results:
    print(doc.page_content)
```

4. RAG 구조에서의 Retriever 역할


[문서 DB] → [Retriever] → [LLM]

1. 문서 DB에서 관련 텍스트를 검색
2. 검색 결과를 LLM 입력 프롬프트에 추가
3. LLM이 문서 기반의 정확한 답변을 생성

이 구조 덕분에 LLM은 훈련 데이터에 없는 최신 지식이나 사내 문서 기반 정보도 활용할 수 있다.

5. Retriever 관련 주요 파라미터

파라미터	설명
k	반환할 문서 개수 (기본 4~5개)
search_type	"similarity", "mmr"(다양성 중심), "similarity_score_threshold" 등
score_threshold	유사도 점수 기준 이하 결과는 제외
filter	메타데이터 필터링 (예: 특정 날짜, 카테고리 등)



6. 정리

항목	설명
역할	질문과 관련 있는 문서를 찾아 LLM에게 전달
핵심 구성요소	벡터스토어, 임베딩 모델, 유사도 검색 알고리즘
장점	외부 지식 연결, 최신 정보 활용, 정확도 향상
활용 예시	FAQ 챗봇, 기업 문서 검색, 논문 요약, 보고서 작성 보조 등



2. VectorStoreRetriever



이 모듈은 RAG 시스템에서 가장 널리 사용되는 기본 검색기(Retriever)입니다.

1. 개념 요약


`VectorStoreRetriever` 는

“벡터 임베딩(Vector Embedding) 기반으로 문서를 검색하는 검색기(Retriever)” 이다.

즉, 문서와 질문을 벡터(숫자 형태) 로 변환해
의미적으로 유사한 문서(청크) 를 찾아주는 역할을 한다.

CSS

[문서] → [임베딩 저장 (Vector DB)] → [질문 임베딩] → [유사도 계산] → [관련 문서 검색]



2. 내부 동작 구조

단계	설명
1. 문서 임베딩 생성	텍스트를 임베딩 모델(예: <code>text-embedding-3-small</code>)로 벡터화
2. 벡터 DB에 저장	FAISS, Chroma, PostgreSQL(pgvector) 등 저장소에 저장
3. 질의 임베딩 생성	사용자의 질문도 같은 모델로 벡터화
4. 유사도 검색(similarity search)	질문 벡터와 문서 벡터 간 거리 계산 (코사인 유사도 등)
5. 결과 반환	가장 유사한 상위 k개의 문서를 반환



3. 기본 코드 예제

python

```
from langchain_community.vectorstores import FAISS
from langchain_openai import OpenAIEmbeddings
from langchain_core.documents import Document

# 1) 문서 준비
docs = [
    Document(page_content="LangChain은 LLM 애플리케이션 개발 프레임워크입니다."),
    Document(page_content="Retriever는 문서에서 관련 정보를 검색합니다."),
    Document(page_content="FAISS는 Facebook AI가 만든 벡터 검색 라이브러리입니다.")
]

# 2) 임베딩 모델
embeddings = OpenAIEmbeddings(model="text-embedding-3-small")

# 3) FAISS 벡터 DB 생성
vectorstore = FAISS.from_documents(docs, embeddings)

# 4) VectorStoreRetriever 생성
retriever = vectorstore.as_retriever(search_kwargs={"k": 2})
```

5) 질의 수행

```
query = "문서를 검색하는 모듈은?"  
results = retriever.invoke(query)
```

6) 결과 출력

```
for i, doc in enumerate(results, 1):  
    print(f"[결과 {i}] {doc.page_content}")
```

출력 예시:

CSS

[결과 1] Retriever는 문서에서 관련 정보를 검색합니다.

[결과 2] LangChain은 LLM 애플리케이션 개발 프레임워크입니다.

4. 주요 매개변수 (`search_kwargs`)

매개변수	설명	예시
<code>k</code>	반환할 문서 개수	<code>{"k": 3}</code>
<code>search_type</code>	검색 방식 ("similarity", "mmr")	<code>{"search_type": "mmr"}</code>
<code>score_threshold</code>	최소 유사도 점수 기준 설정	<code>{"score_threshold": 0.8}</code>
<code>filter</code>	메타데이터 기반 필터링	<code>{"filter": {"category": "finance"}}</code>

5. `search_type` 옵션

타입	설명
<code>"similarity"</code>	단순히 유사도가 높은 문서 k개 반환
<code>"mmr"</code>	MMR(Maximal Marginal Relevance) 방식으로 다양성 확보 (중복된 내용 방지)


MMR 예시:

"LangChain 개요"와 "LangChain 설치 방법" 둘 다 포함되게 검색 결과를 다양화.



6. VectorStoreRetriever의 장점

장점	설명
빠름	벡터 기반 거리 계산으로 대규모 문서 검색 가능
정확함	단어 일치가 아니라 의미적 유사성으로 검색
유연함	다양한 벡터 스토어(FAISS, Chroma, Milvus, PostgreSQL 등) 지원
통합 쉬움	<code>.as_retriever()</code> 로 즉시 변환 가능 (RAG 체인에 쉽게 연결)



7. 실제 사용 구조 (RAG 연결 예시)

▼ 1. 필수 모듈 임포트

```
from langchain_community.vectorstores import FAISS
from langchain_openai import OpenAIEmbeddings, ChatOpenAI
from langchain_core.documents import Document
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser
from langchain_core.runnables import RunnableParallel, RunnablePassthrough
```

▼ 2. 문서 준비 (예시용)

```
docs = [
    Document(page_content="LangChain은 LLM 애플리케이션 개발 프레임워크이다."),
    Document(page_content="FAISS는 Facebook AI가 만든 벡터 검색 라이브러리이다."),
    Document(page_content="Retriever는 질문과 유사한 문서를 찾아주는 역할을 한다."),
]
```

▼ 3. 임베딩 모델 선택

```
embeddings = OpenAIEmbeddings(model="text-embedding-3-small")
```

▼ 4. VectorStore (FAISS) 생성

```
vectorstore = FAISS.from_documents(docs, embeddings)
```

▼ 5. VectorStoreRetriever 생성

```
retriever = vectorstore.as_retriever(search_kwargs={"k": 2})
```

▼ 6. 프롬프트 정의

```
prompt = ChatPromptTemplate.from_template("""  
아래 문서를 참고하여 질문에 답하세요.
```

문서 내용:

```
{context}
```

질문:

```
{question}  
""")
```

▼ 7. LLM 및 파서 설정

```
llm = ChatOpenAI(model="gpt-4o-mini")  
parser = StrOutputParser()
```

▼ 8. RAG 체인 구성


```
rag_chain = (  
    RunnableParallel({"context": retriever, "question": RunnablePassthrough()})  
    | prompt  
    | llm  
    | parser  
)
```

▼ 9. 질의 실행

```
query = "FAISS는 무엇을 하는 라이브러리인가요?"  
response = rag_chain.invoke(query)
```

▼ 10. 결과 출력

```
print("질문:", query)  
print("답변:", response)
```

8. 정리

항목	설명
이름	VectorStoreRetriever
역할	벡터 기반 의미 검색 수행
입력	질의문(자연어)
출력	유사 문서 목록
주요 사용처	RAG, 문서 기반 챗봇, FAQ 시스템, 사내 문서 검색
대표 백엔드	FAISS, Chroma, PostgreSQL(pgvector), Milvus 등



2.1 MMR(Maximal Marginal Relevance) 검색

MMR(Maximal Marginal Relevance) 검색은

LangChain의 `VectorStoreRetriever` 등에서 검색 결과의 “다양성(Diversity)”을 높이기 위한 검색 방식입니다.

✓ 1. 기본 개념

일반적인 벡터 검색(`similarity`)은

“질문과 가장 유사한 문서 k개”만 단순히 반환

하지만 이 방식은 종종

비슷한 내용의 문서만 여러 개 나와서 **중복된 결과**가 많습니다.

그래서 나온 방법이 **MMR (Maximal Marginal Relevance)** 입니다.

✅ 2. 핵심 아이디어

MMR은 유사성(Similarity) 과 다양성(Diversity) 의 균형을 맞춥니다.

즉,

“질문과 유사하면서도, 이미 선택된 문서들과는 겹치지 않게 하라!”

라는 전략입니다.

이를 수식으로 표현하면 🗣️

$$\text{MMR} = \arg \max_{D_i \in R \setminus S} [\lambda \cdot \text{Sim}(D_i, Q) - (1 - \lambda) \cdot \max_{D_j \in S} \text{Sim}(D_i, D_j)]$$

기호	의미
D_i	후보 문서
Q	질문 벡터
R	전체 문서 집합
S	이미 선택된 문서 집합
λ	유사성 vs 다양성 비율 (0~1 사이)
<code>Sim()</code>	코사인 유사도 등 벡터 유사도 계산 함수

✅ 3. 간단한 예시

질문: "LangChain Retriever의 역할은?"

검색 결과 후보 문서:

문서	내용	유사도
A	"Retriever는 문서를 검색한다."	0.95
B	"LangChain은 LLM 프레임워크이다."	0.92
C	"Retriever는 RAG 구조의 핵심이다."	0.91
D	"FAISS는 빠른 벡터 검색 라이브러리이다."	0.88

일반 `similarity` 검색 → A, B, C 반환

→ 내용이 거의 비슷 (중복 많음)

MMR 검색 → A (가장 유사), D (다양성 확보), C (적절한 보완)

→ 더 폭넓은 정보 포함

✓ 4. LangChain에서 사용하는 방법

python

```
retriever = vectorstore.as_retriever(  
    search_type="mmr",  
    search_kwargs={"k": 3, "lambda_mult": 0.5}  
)
```

옵션

설명

`search_type="mmr"`

MMR 검색 활성화

`k`

반환할 문서 수

`lambda_mult`

유사성과 다양성의 비율 (기본 0.5, 높을수록 유사성 우선)

✓ 5. 요약 정리

항목	설명
목적	중복된 문서를 줄이고 다양한 관점의 결과 제공
기본 원리	"질문과의 유사성" - "이미 선택된 문서와의 중복도"
조절 파라미터	<code>lambda_mult</code> (0~1 사이 비율)
장점	중복 감소, 정보 다양성 향상
단점	약간의 검색 속도 저하 (계산량 증가)

✦ 한 줄 요약:

MMR 검색은 "질문과 비슷하면서도 서로 다른 내용의 문서"를 골라주는,
중복 최소화 + 다양성 보장형 벡터 검색 방식이다.

3. 다중 쿼리 검색기 (Multi Query Retriever)

LangChain에서 “질문을 여러 방식으로 바꿔서 검색 정확도를 높이는” 아주 강력한 Retriever입니다.

✓ 1. 기본 개념

`MultiQueryRetriever` 는 하나의 질문을 받아 LLM을 사용해 다양한 표현(패러프레이즈) 으로 여러 개의 질의문을 생성하고, 그 각각을 벡터 검색기에 보내서 더 풍부한 관련 문서를 찾는 Retriever 입니다.

즉,

“사용자 질문을 여러 버전으로 다시 써서, 검색 누락을 최소화하는 Retriever”

입니다.

✅ 2. 왜 필요한가?

기본 `VectorStoreRetriever` 는 하나의 질의 벡터만 사용합니다.

→ 질문이 조금만 다르게 표현돼도 중요한 문서를 놓칠 수 있음.

예를 들어:

질문: "LangChain의 Retriever 구조는?"

하지만 문서에는 "문서 검색기 구조" 라고 적혀 있다면?

단일 질의 검색은 이 문서를 찾지 못할 수도 있습니다.

이럴 때 `MultiQueryRetriever`가 다음과 같은 질의를 자동으로 만들어냅니다:

- "LangChain에서 문서 검색기 구조는?"
- "Retriever가 하는 일은?"
- "LangChain의 검색 시스템은 어떻게 동작하는가?"

이렇게 여러 질의로 검색해 **정확도(Recall)** 를 높입니다.

✓ 3. 동작 구조

SCSS

사용자 질문



LLM → 질문 3~5개 생성



각 질문으로 벡터 검색 수행



검색 결과 통합 (중복 제거)



최종 관련 문서 목록 반환

✦ 한 줄 요약:

`MultiQueryRetriever` 는 하나의 질문을 LLM이 여러 방식으로 다시 써서 검색하여,
“놓치는 문서 없이 더 풍부한 결과” 를 찾아주는 고급 검색기이다.

✓ 4. 코드 예제

python

```
from langchain_community.vectorstores import FAISS
from langchain_openai import ChatOpenAI, OpenAIEmbeddings
from langchain.retrievers.multi_query import MultiQueryRetriever
from langchain_core.documents import Document

# 1. 문서 데이터
docs = [
    Document(page_content="LangChain은 LLM 애플리케이션 개발 프레임워크이다."),
    Document(page_content="Retriever는 문서에서 관련 정보를 검색한다."),
    Document(page_content="FAISS는 빠른 벡터 검색 라이브러리이다."),
    Document(page_content="LangChain의 검색 시스템은 Retriever를 사용한다."),
]

# 2. 벡터스토어 생성
embeddings = OpenAIEmbeddings(model="text-embedding-3-small")
vectorstore = FAISS.from_documents(docs, embeddings)
```

3. LLM 및 기본 검색기

```
llm = ChatOpenAI(model="gpt-4o-mini")
```

```
base_retriever = vectorstore.as_retriever(search_kwargs={"k": 2})
```

4. MultiQueryRetriever 생성

```
multi_retriever = MultiQueryRetriever.from_llm(  
    retriever=base_retriever,  
    llm=llm  
)
```

5. 질의 실행

```
query = "LangChain의 문서 검색 구조는?"
```

```
results = multi_retriever.invoke(query)
```

6. 결과 출력

```
for i, doc in enumerate(results, 1):  
    print(f"[결과 {i}] {doc.page_content}")
```

✓ 5. 동작 예시

LLM이 자동으로 생성한 질의 (예시):

- 1) LangChain의 검색 시스템은 어떻게 구성되어 있나요?
- 2) Retriever의 구조는 무엇인가요?
- 3) LangChain에서 문서를 검색하는 방식은?

→ 각 질의별로 벡터 검색 수행

→ 결과를 합쳐 중복 제거 후 반환

결과:

CSS

[결과 1] Retriever는 문서에서 관련 정보를 검색한다.

[결과 2] LangChain의 검색 시스템은 Retriever를 사용한다.

✓ 6. 장점

장점

설명

검색 누락 최소화

질문의 다양한 표현을 통해 관련 문서 탐색률(Recall) 향상

자연스러운 질의 변환

LLM이 의미 기반으로 질의를 재작성

자동화

사용자는 단일 질문만 입력하면 됨

✓ 7. 단점

단점

설명

LLM 호출 비용 증가

다중 질의 생성을 위해 1회 LLM 호출 필요

중복 결과 발생 가능

유사 문서가 여러 질의에 포함될 수 있음 (내부 중복 제거로 보정)

속도 저하

질의 수가 많을수록 검색 횟수 증가

✓ 8. 주요 파라미터

파라미터	설명
<code>retriever</code>	기본 검색기 (예: VectorStoreRetriever)
<code>llm</code>	질의 생성에 사용할 언어 모델
<code>prompt</code>	(선택) 질의 재작성용 프롬프트 템플릿
<code>num_queries</code>	(선택) 생성할 질의 개수 (기본 3개)

✓ 9. 요약 정리

항목	설명
이름	MultiQueryRetriever
핵심 기능	질문을 여러 버전으로 변환해 다양한 각도에서 검색
LLM 사용 여부	✓ (질의 재작성에 사용)
장점	검색 누락 방지, Recall 향상
단점	속도·비용 증가
추천 사용 상황	문서 표현이 다양한 대규모 데이터셋 (논문, 보고서, FAQ 등)

4. 문맥 압축 검색기 (ContextualCompressionRetriever)

이 모듈은 RAG의 검색 결과를 “요약·압축”해서 전달함으로써,
LLM이 더 작은 입력으로도 중요한 정보만 이해할 수 있게 해주는 Retriever 입니다.

✓ 1. 기본 개념

`ContextualCompressionRetriever` 는

일반 Retriever(예: `VectorStoreRetriever`)가 가져온 문서들 중
“핵심 내용만 추출(요약)”해서 LLM에게 전달하는 역할을 합니다.

즉,

“많은 문서를 그대로 전달하지 않고, 질문과 관련된 문맥만 압축해서 전달한다.”

라는 개념이에요.

✓ 2. 동작 구조

아래는 `ContextualCompressionRetriever` 의 처리 흐름입니다:

markdown

사용자 질문



기본 Retriever (예: VectorStoreRetriever)



검색된 문서 5개



문맥 압축기 (LLM 기반 Compressor)




질문과 관련된 문장/문단만 남김



LLM으로 전달 → 최종 답변 생성

즉, "검색 + 압축(요약)"의 2단계 구조입니다.

LLM이 처리할 컨텍스트 길이를 크게 줄이면서도 정보 손실을 최소화할 수 있습니다.



✓ 3. 주요 구성요소

구성요소	설명
base_retriever	기본 검색기 (<code>VectorStoreRetriever</code> , <code>MultiQueryRetriever</code> 등)
compressor	문서를 요약·압축하는 컴포넌트 (LLM 기반)

LangChain은 이 두 컴포넌트를 결합해서 `ContextualCompressionRetriever` 객체를 만듭니다.



✓ 4. 간단한 코드 예제

python

📄 코드 복사

```
from langchain_community.vectorstores import FAISS
from langchain_openai import OpenAIEmbeddings, ChatOpenAI
from langchain.retrievers import ContextualCompressionRetriever
from langchain.retrievers.document_compressors import LLMChainExtractor
from langchain_core.documents import Document

# 1. 기본 문서
docs = [
    Document(page_content="LangChain은 LLM 애플리케이션 개발 프레임워크입니다."),
    Document(page_content="Retriever는 문서에서 관련 정보를 검색합니다."),
    Document(page_content="ContextualCompressionRetriever는 검색된 문서를 요약해 전달합니다.")
]

# 2. 벡터스토어 생성
embeddings = OpenAIEmbeddings(model="text-embedding-3-small")
vectorstore = FAISS.from_documents(docs, embeddings)
base_retriever = vectorstore.as_retriever(search_kwargs={"k": 3})
```

3. LLM 기반 요약 압축기 생성

```
llm = ChatOpenAI(model="gpt-4o-mini")  
compressor = LLMChainExtractor.from_llm(llm)
```

4. ContextualCompressionRetriever 생성

```
retriever = ContextualCompressionRetriever(  
    base_compressor=compressor,  
    base_retriever=base_retriever  
)
```

5. 질의 실행

```
query = "문서 검색 결과를 요약해주는 Retriever는?"  
results = retriever.invoke(query)
```

6. 결과 출력

```
for i, doc in enumerate(results, 1):  
    print(f"[요약 {i}] {doc.page_content}")
```

✓ 5. 동작 예시

단계	내용
일반 Retriever	"LangChain은 LLM 프레임워크입니다.", "Retriever는 문서를 검색합니다.", "ContextualCompressionRetriever는 검색된 문서를 요약해 전달합니다."
압축 후 결과	"ContextualCompressionRetriever는 검색된 문서를 요약해 전달합니다."

즉, 단순 검색 결과 중 질문과 가장 직접적으로 관련된 문장만 남겨줍니다.

✓ 6. 장점

장점	설명
컨텍스트 절약	긴 문서를 그대로 넣지 않고 핵심 문맥만 전달
응답 품질 향상	질문과 직접 관련된 부분만 LLM이 보게 됨
RAG 효율 향상	입력 토큰 수 절감 → 속도/비용 감소

✅ 7. 단점

단점	설명
추가 LLM 호출 비용	압축 과정에서 한 번 더 LLM을 사용
요약 품질 의존도	압축 품질은 LLM의 성능에 따라 달라짐

✅ 8. 요약 정리

항목	내용
이름	ContextualCompressionRetriever
역할	검색 결과를 요약·압축하여 전달
구성요소	base_retriever + compressor
대표 압축기	<code>LLMChainExtractor</code> , <code>EmbeddingsFilter</code>
사용 목적	RAG 파이프라인에서 컨텍스트 최소화 및 품질 향상
장점	효율적이고 집중도 높은 검색 결과 제공



감사합니다