생성형 AI 기반 RAG 개발자 과정

ChromaDB 구축
영속화 · 메타데이터 · 검색

Session 14 / 33 — Day 2

persist · collection · metadata filter · 중복방지 · 재색인

박수현 · (사)한국정보공학기술사회 · 2026

LLM 이론 · LangChain · ChromaDB CRUD · Agent — 4일 핸즈온

이번 시간을 마치면 — 원리

원리 (본선)

- ✅ **영속화가 필요한 이유** — InMemory vs Chroma (재임베딩 비용 0)
- ✅ ChromaDB **저장 구조** — 컬렉션 {id·embedding·document·metadata} + HNSW + persist
- ✅ **collection / persist / metadata filter** 의 의미
- ✅ chunk **deterministic ID** = 중복 방지의 핵심

원리 (심화)

- ✅ Vector DB 6종 비교 · 백엔드(SQLite·Parquet) 내부 · 재색인 비용 분석

> 📂 소스: `2.langchain/7.RAG/3.vectorstore/`

이번 시간을 마치면 — 실습

실습

- ✅ ChromaDB Persistent client + collection CRUD
- ✅ **PDF 멀티로딩** — 5개 PDF 통째로 색인 + 페이지별 메타데이터
- ✅ **메타데이터 필터** — `$eq` / `$gt` / `$in` / `$and` 문법으로 정밀 검색
- ✅ 한국어 임베딩 모델 교체 (multilingual-e5)
- ✅ 내 문서 폴더 — `~/documents/` 통째로 색인

> 📂 소스: `2.langchain/7.RAG/3.vectorstore/`

PART A

PART A
Chroma 와 영속화

> FAISS 의 한계와 디스크 영속화의 가치 — 약 15분
>
> 🔗 벡터 DB 4종 비교·ANN 색인 구조는 **Session 13(PART A·C)** 소유. 여기서는 그중 **Chroma 의 영속화·컬렉션·필터 구조**에 집중한다.

Session 13 FAISS 의 한계

한계	증상	운영 영향
메모리 기반	프로세스 죽으면 색인 사라짐 (`save_local()` 매번 직렬화)	재시작마다 로드, 분산 불가
메타데이터 필터 약함	LangChain 래퍼가 retrieval 후 후처리로만 필터	"특정 PDF 의 12 페이지만 검색" 불편
동시 쓰기 X	단일 프로세스 mutation	색인 중 검색 불가
하이브리드 검색 X	벡터만 — BM25 (키워드) 결합 어려움	약어·고유명사 검색 부정확

FAISS 가 적합한 경우

- **학습·실험·소규모** (< 100만 벡터)
- 색인을 한 번만 만들고 그대로 쓰는 경우
- 응답 속도가 절대적으로 우선 (raw FAISS HNSW: < 1ms)

한 단계 위의 벡터 DB 가 필요한 경우

- 문서가 **추가·삭제·갱신** 됨
- **메타데이터 필터** 가 검색의 일부
- **여러 사용자·프로세스** 가 동시 접근
- **출처 인용**과 함께 답변해야 함

재실행마다 재임베딩 — 누적되는 비용

InMemory 의 문제

- 메모리 위에서만 동작 → **프로세스가 끝나면 임베딩이 사라짐**
- 스크립트를 다시 실행할 때마다 **모든 청크를 임베딩 API 로 재계산**
- 청크가 많아질수록 시간·비용이 누적

Chroma 의 해결

- 임베딩을 `persist_directory` 폴더에 **디스크 저장**
- 재실행 시 **즉시 로드** → 임베딩 API 재호출 0
- 웹 서버가 재시작돼도 인덱스 유지

결정적 차이

구분	InMemory	Chroma
저장 위치	메모리	`persist_directory` (디스크)
재실행 시	매번 재임베딩, API 비용	**즉시 로드, 추가 비용 0**
용도	빠른 데모	반복 실험·운영

```
# 있으면 로드, 없으면 생성 — 비용을 아끼는 핵심 패턴
if os.path.exists(PERSIST_DIR) and os.listdir(PERSIST_DIR):
    store = load_store()    # 재임베딩 없음
else:
    store = build_store()   # 첫 1회만 임베딩
```

> 🎯 InMemory 와의 결정적 차이. Chroma 를 채택하는 가장 직접적인 이유는 영속화를 통한 임베딩 비용 0 이다.

시장 주요 옵션 비교

DB	호스팅	메타데이터 필터	BM25 하이브리드	가격	적합
FAISS	라이브러리	약함	❌	무료	학습·실험
ChromaDB ⭐	로컬 + 클라이언트-서버	**강함**	❌ (외부 BM25 결합)	무료	**본 코스** · 개인 RAG
Qdrant	로컬 + 클라우드	강함	✅	무료 + 호스팅	단일 회사 운영
Weaviate	로컬 + 클라우드	강함	✅	무료 + 호스팅	멀티모달·하이브리드
Pinecone	**클라우드 전용**	강함	✅	유료 (월 $70+)	SaaS 운영
Milvus	클러스터	강함	✅	무료	대규모 (수억+)
pgvector	PostgreSQL 확장	**SQL**	✅ (Postgres FTS)	무료	이미 Postgres 쓰는 팀

선택 가이드 (2026)

1. **개인·학습** → FAISS or **Chroma**
2. **소~중규모 운영** (< 1억) → **Chroma server / Qdrant**
3. **대규모·멀티모달** → Weaviate / Milvus
4. **이미 RDB 쓰는 팀** → **pgvector** (인프라 절약)
5. **빠른 SaaS 출시** → Pinecone (관리 부담 ↓, 비용 ↑)

내부 동작 구조

핵심 컴포넌트

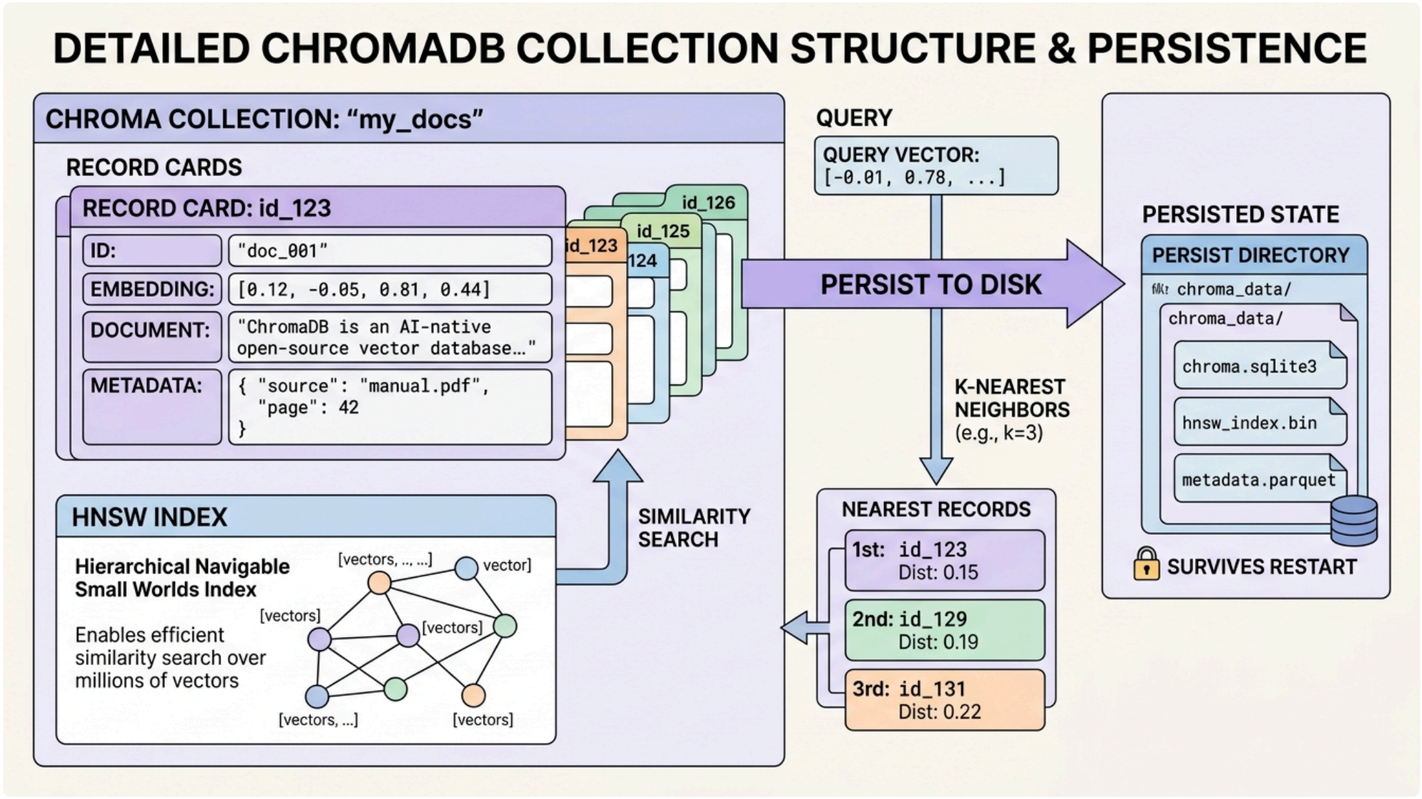
레이어	구현	역할
Client API	Python / TypeScript	`add / query / get / update / delete`
Embedding Function	OpenAI / SentenceTransformers / 커스텀	텍스트 → 벡터
Index	**HNSW** (Session 13 동일 구조)	벡터 → top-K 검색
Storage	**SQLite + Parquet**	메타데이터 · 문서 본문 · 임베딩
Persistence	local dir 또는 Server (FastAPI)	`persist_directory` 지정

데이터 흐름

```
add(documents)
   ├⟶ Embedding Function 호출 → 벡터
   ├⟶ HNSW 인덱스에 벡터 + ID
   └⟶ SQLite + Parquet 에 (id, document, metadata, embedding) 저장

query(query_text, where={...})
   ├⟶ Embedding Function 호출 → 질의 벡터
   ├⟶ HNSW 에서 top-K 후보
   ├⟶ SQLite 에서 where 조건 일치하는 것만 필터
   └⟶ (ids, distances, documents, metadatas) 반환
```

컬렉션 · 레코드 · 인덱스



ChromaDB 컬렉션 · 레코드 {id·embedding·document·metadata}, HNSW 인덱스로 similarity_search, persist_directory 영속화

PART B

PART B
ChromaDB 첫 만남

> 설치 → CRUD → 검색 — 약 20분

두 가지 모드 — Persistent (로컬 파일)

```
# 본 코스 (개인 RAG)
pip install chromadb langchain-chroma
```

Persistent (로컬 파일)

```
import chromadb

# 로컬 디렉토리에 저장
client = chromadb.PersistentClient(path="./chroma_db")
# 디렉토리 구조:
# ./chroma_db/
#   ├ chroma.sqlite3        ← collection 메타·문서
#   └ <uuid>/                ← collection 별 HNSW + parquet
```

> 🎯 본 코스는 **Persistent** 로 진행한다. 운영 환경에서는 server 모드 + 별도 Docker 를 사용한다.

두 가지 모드 — HttpClient / 메모리

HttpClient (서버 모드 — 운영용)

```
# 서버 실행 (별도 터미널)
chroma run --path ./chroma_db --host 0.0.0.0 --port 8000
```

```
client = chromadb.HttpClient(host="localhost", port=8000)
```

메모리 모드 (단위 테스트용)

```
client = chromadb.Client()   # 프로세스 종료 시 사라짐
```

> 🎯 운영 환경에서는 HttpClient + 별도 Docker 컨테이너로 분리한다.

C·R — 생성 / 추가 / 조회

```
import chromadb

client = chromadb.PersistentClient(path="./chroma_db")

# ═══ Collection 생성 / 가져오기 ===
collection = client.get_or_create_collection(
    name="my_notes",
    metadata={"hnsw:space": "cosine"},   # 거리 함수 (cosine|l2|ip)
)

# === C — add ═══
collection.add(
    ids=["doc1_p1", "doc1_p2", "doc1_p3"],
    documents=[
        "RAG 는 retrieval augmented generation 의 약자다.",
        "벡터 DB 는 임베딩을 저장하고 유사도 검색을 한다.",
        "ChromaDB 는 SQLite + Parquet 으로 영속화한다.",
    ],
    metadatas=[
        {"source": "rag_intro.md", "page": 1},
        {"source": "rag_intro.md", "page": 2},
        {"source": "rag_intro.md", "page": 3},
    ],
)
```

⬇️ 코드가 길어 다음 장에 이어집니다.

C·R — 생성 / 추가 / 조회 (이어서)

```
# ═══ R — query (유사도 검색) ═══
results = collection.query(
    query_texts=["벡터 검색이 뭔가요?"],
    n_results=2,
)
# results = {ids, distances, documents, metadatas, embeddings}

# ═══ R — get (ID 또는 조건으로) — 검색 아닌 단순 조회 ===
collection.get(ids=["doc1_p1"])
collection.get(where={"source": "rag_intro.md"})
collection.count()                          # 저장된 청크 수 — 색인 검증의 첫걸음
collection.get(include=["documents", "metadatas"], limit=3)  # 내용·메타 직접 확인
```

> 💡 `ids` 는 client 가 제공 — auto-generated 가 아님. 중복 방지·재색인의 핵심 (뒤 슬라이드).

U·D — 갱신 / 삭제

```
# ═══ U — update (기존 ID 의 내용·메타 교체) ═══
collection.update(
    ids=["doc1_p1"],
    documents=["RAG = 검색을 통해 LLM 응답을 보강하는 패턴"],
)

# ═══ D — delete (ID 또는 조건으로) ═══
collection.delete(ids=["doc1_p3"])
collection.delete(where={"source": "old_doc.md"})
```

5가지 동사 한눈에

동사	메서드	용도
C	`add`	청크 + 메타 + ID 삽입
R	`query`	유사도 검색 (벡터)
R	`get` / `count`	ID·조건 조회, 색인 검증
U	`update`	기존 ID 내용 교체
D	`delete`	ID·조건 삭제

> 💡 `update` 는 **같은 ID 가 이미 있어야** 동작. 없는 ID 는 무시 → 신규 삽입은 `add` / `upsert` 사용.

임베딩 함수 3가지 옵션

```
import chromadb
from chromadb.utils import embedding_functions

# ═══ 1. Sentence Transformers (기본, 영어 위주) ═══
default_ef = embedding_functions.SentenceTransformerEmbeddingFunction(
    model_name="all-MiniLM-L6-v2"   # 384d, 빠름, 무료
)

# ═══ 2. OpenAI (1536d, 한국어 우수) ═══
openai_ef = embedding_functions.OpenAIEmbeddingFunction(
    api_key=os.environ["OPENAI_API_KEY"],
    model_name="text-embedding-3-small",
)

# ═══ 3. 한국어 특화 — multilingual-e5-large ═══
multilingual_ef = embedding_functions.SentenceTransformerEmbeddingFunction(
    model_name="intfloat/multilingual-e5-large"   # 1024d, 100+ 언어
)

# Collection 생성 시 임베딩 함수 지정
collection = client.get_or_create_collection(
    name="ko_notes",
    embedding_function=multilingual_ef,   # ← 한국어면 이거 권장
)
```

⚠️ **임베딩 함수는 collection 생성 후 변경 불가**. 모델 교체 시 collection 재생성 + 재색인이 필요하다.

임베딩 모델 한국어 비교

한국어 벤치마크 (개략 평가)

모델	차원	한국어 품질	속도	가격
`all-MiniLM-L6-v2`	384	❌ 약함	⚡⚡⚡	무료
`multilingual-e5-large`	1024	✅ 우수	⚡⚡	무료 (로컬)
OpenAI text-embedding-3-small	1536	✅ 우수	⚡⚡⚡ (API)	$0.02/M 토큰
`BAAI/bge-m3`	1024	✅ 최상위	⚡⚡	무료 (로컬)

> ⚠️ **임베딩 함수는 collection 생성 후 변경 불가**. 모델 교체 시 collection 재생성 + 재색인이 필요하다.

`langchain-chroma` 추상화

```
from langchain_chroma import Chroma
from langchain_openai import OpenAIEmbeddings
from langchain_core.documents import Document

# 1. 첫 색인 (디렉토리에 저장)
docs = [
    Document(page_content="...", metadata={"source": "doc1.md", "page": 1}),
    # ...
]

vectorstore = Chroma.from_documents(
    documents=docs,
    embedding=OpenAIEmbeddings(model="text-embedding-3-small"),
    persist_directory="./chroma_db",
    collection_name="my_notes",
)
```

⬇️ 코드가 길어 다음 장에 이어집니다.

`langchain-chroma` 추상화 (이어서)

```
# 2. 다음번 — 기존 디렉토리에서 로드 (재임베딩 X, API 비용 0)
vectorstore = Chroma(
    persist_directory="./chroma_db",
    collection_name="my_notes",
    embedding_function=OpenAIEmbeddings(model="text-embedding-3-small"),
)
# "있으면 로드, 없으면 생성" 패턴:
#   from_documents = 임베딩 후 저장 (첫 1회) / Chroma(...) = 재임베딩 없는 로드

# 3. 추가 — 점진 색인 (기존에 더 얹기)
vectorstore.add_documents(more_docs)

# 4. 검색 + 메타데이터 필터
results = vectorstore.similarity_search(
    "벡터 검색이란?",
    k=4,
    filter={"source": "rag_intro.md"},   # 특정 문서로 한정
)

# 5. retriever 추상화 (Session 13 의 RAG 체인과 동일)
```

🎯 FAISS → Chroma 마이그레이션은 **3줄 변경으로 충분** (`FAISS` → `Chroma` + `persist_directory` + `collection_name`).

💡 `persist_directory` **가 핵심**: InMemory 는 프로세스 종료 시 사라져 매 실행 재임베딩(비용↑). Chroma 는 디스크에 영속화 → 재실행 시 즉시 로드, 임베딩 API 재호출 0.

PART C

PART C
메타데이터 필터 — 정밀 검색의 핵심

> $eq · $gt · $in · $and — 약 12분

ChromaDB `where` 절 문법 정리

연산자	의미	예시
`$eq` (기본)	정확히 같음	`{"page": 12}` 또는 `{"page": {"$eq": 12}}`
`$ne`	같지 않음	`{"source": {"$ne": "draft.md"}}`
`$gt` / `$gte`	보다 큼 / 이상	`{"page": {"$gte": 10}}`
`$lt` / `$lte`	보다 작음 / 이하	`{"page": {"$lt": 50}}`
`$in`	리스트 중 하나	`{"category": {"$in": ["AI", "ML"]}}`
`$nin`	리스트 중 어느 것도 아님	`{"status": {"$nin": ["draft", "deprecated"]}}`
`$and`	모두 만족	`{"$and": [{"src": "x.pdf"}, {"page": {"$gte": 10}}]}`
`$or`	하나만 만족	`{"$or": [{"src": "a.pdf"}, {"src": "b.pdf"}]}`
`$contains` (document)	본문 substring	`vectorstore.similarity_search(q, filter={...}, where_document={"$contains":"강아지"})`

사용 위치 2종

- `where` — metadata 필터
- `where_document` — 문서 본문 substring 필터 (검색 hybrid 효과)

운영 RAG 에서 자주 쓰는 패턴

```
# 1. 특정 PDF 만
vectorstore.similarity_search("Q", filter={"source": "manual.pdf"})

# 2. 페이지 범위 — Chapter 3 (page 30~50)
vectorstore.similarity_search("Q", filter={
    "$and": [
        {"source": "book.pdf"},
        {"page": {"$gte": 30}},
        {"page": {"$lte": 50}},
    ]
})

# 3. 여러 출처 — A 또는 B
vectorstore.similarity_search("Q", filter={
    "source": {"$in": ["doc_a.md", "doc_b.md", "doc_c.md"]}
})

# 4. draft 제외
vectorstore.similarity_search("Q", filter={
    "status": {"$ne": "draft"}
})
```

⬇️ 코드가 길어 다음 장에 이어집니다.

운영 RAG 에서 자주 쓰는 패턴 (이어서)

```
# 5. 최근 1개월 색인된 문서만
import time
one_month_ago = time.time() - 30 * 86400
vectorstore.similarity_search("Q", filter={
    "embedded_at_ts": {"$gte": one_month_ago}
})

# 6. 사용자별 격리 (멀티테넌트)
vectorstore.similarity_search("Q", filter={"user_id": current_user})

# 7. 본문 + 메타 결합 — "강아지" 단어가 들어간 manual.pdf chunk
vectorstore.similarity_search(
    "Q",
    filter={"source": "manual.pdf"},
    where_document={"$contains": "강아지"},
)
```

> 🎯 **필터는 retrieval 정확도를 2~3배 향상시킨다**. 답변 근거의 출처 범위를 검색 단계에서 지정하는 수단이다.

PART D

PART D
PDF 멀티로딩 — 문서 컬렉션 구축

> 5개 PDF 통째로 색인 — 약 20분

한 PDF → Document 리스트

```
from langchain_community.document_loaders import PyPDFLoader

loader = PyPDFLoader("./DATA/Python_시큐어코딩_가이드(2023).pdf")
pages = loader.load()                          # 페이지마다 Document 1개

print(f"총 페이지: {len(pages)}")             # 예: 87
print(f"3페이지 본문 일부: {pages[2].page_content[:200]}")
print(f"3페이지 메타데이터: {pages[2].metadata}")
# {'source': './DATA/Python_시큐어코딩_가이드.pdf', 'page': 2}
```

한국어 PDF 주의 4가지

이슈	증상	해결
스캔 이미지 PDF	`page_content` 가 빈 문자열	OCR 필요 — `easyocr` 또는 `pytesseract`
암호화 PDF	로드 자체 실패	`pypdf.PdfReader(pwd=...)`
CID 인코딩	`(cid:1234)` 같은 문자열	`pdfplumber` 또는 `pdfminer.six` 로 폴백
글자 깨짐	한글이 □□□ 로	폰트 임베딩 안 됨 → OCR 필요

> 💡 본 코스 권장: `PyPDFLoader` 가 정상 동작하면 그대로 사용하고, 본문이 깨지면 `UnstructuredFileLoader` 또는 `pdfplumber` 로 폴백한다.

`pdfloader.py` 핵심 — 페이지 단위 + 청킹

```
from langchain_community.document_loaders import PyPDFLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_chroma import Chroma
from langchain_openai import OpenAIEmbeddings
from pathlib import Path
import hashlib

pdf_path = Path("./DATA/Python_시큐어코딩_가이드.pdf")
loader = PyPDFLoader(str(pdf_path))
pages = loader.load()

# 청킹 — 토큰 단위 (비용 추정 정확)
splitter = RecursiveCharacterTextSplitter.from_tiktoken_encoder(
    separator="\n\n",
    chunk_size=1000,
    chunk_overlap=200,
)
chunks = splitter.split_documents(pages)
```

⬇️ 코드가 길어 다음 장에 이어집니다.

`pdfloader.py` 핵심 — 페이지 단위 + 청킹 (이어서)

```
# 메타데이터 풍부화
for i, c in enumerate(chunks):
    c.metadata["doc_id"] = pdf_path.stem               # 파일명만
    c.metadata["chunk_idx"] = i
    c.metadata["chunk_total"] = len(chunks)

    # 결정적 ID — 같은 내용은 항상 같은 ID
    h = hashlib.sha256(c.page_content.encode()).hexdigest()[:16]
    c.metadata["chunk_id"] = f"{pdf_path.stem}_p{c.metadata['page']}_h{h}"

# 색인
vectorstore = Chroma.from_documents(
    documents=chunks,
    embedding=OpenAIEmbeddings(model="text-embedding-3-small"),
    persist_directory="./chroma_db",
    collection_name="my_library",
    ids=[c.metadata["chunk_id"] for c in chunks],   # ← 결정적 ID 전달!
)
print(f"{len(chunks)} chunks 색인 완료")
```

결정적 ID 의 효용

- **같은 PDF 재색인 시 중복 방지** (내용 같은 chunk = 같은 ID = upsert)
- **부분 수정 시 정확한 삭제** (`collection.delete(ids=[...])`)

`~/documents/*.pdf` 한 번에

```
from pathlib import Path
from langchain_community.document_loaders import PyPDFLoader, DirectoryLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_chroma import Chroma
from langchain_openai import OpenAIEmbeddings

# ═══ 옵션 1 — DirectoryLoader (LangChain 추상화) ═══
loader = DirectoryLoader(
    path="./my_library/",
    glob="**/*.pdf",
    loader_cls=PyPDFLoader,
    use_multithreading=True,
    show_progress=True,
)
docs = loader.load()
print(f"PDF 합계 페이지: {len(docs)}")
```

💡 가장 간단한 일괄 로딩 방식. 메타데이터를 세밀히 제어하려면 다음 장의 **옵션 2** 를 사용한다.

`~/documents/*.pdf` 한 번에 — 직접 제어

```
# === 옵션 2 — 직접 (더 세밀한 제어) ===
all_chunks = []
splitter = RecursiveCharacterTextSplitter.from_tiktoken_encoder(
    chunk_size=1000, chunk_overlap=200
)

for pdf in sorted(Path("./my_library").glob("**/*.pdf")):
    pages = PyPDFLoader(str(pdf)).load()
    for p in pages:
        p.metadata["doc_id"] = pdf.stem
        p.metadata["doc_path"] = str(pdf.relative_to("./my_library"))
    chunks = splitter.split_documents(pages)
    all_chunks.extend(chunks)
    print(f"  ✓ {pdf.name}: {len(pages)} pages → {len(chunks)} chunks")

# 한 collection 에 통째 색인
vectorstore = Chroma.from_documents(
    documents=all_chunks,
    embedding=OpenAIEmbeddings(model="text-embedding-3-small"),
    persist_directory="./chroma_db",
    collection_name="library",
)
print(f"총 {len(all_chunks)} chunks 색인 완료")
```

> 💡 **5만 chunks 수준까지** 는 노트북에서 무리 없이 처리된다. 그 이상은 색인 시간(~30분)과 비용($1~$5)을 고려한다.

2.langchain/7.RAG/3.vectorstore/3.chromadb2_saveload_multifil 패턴

```
# 이미 색인된 vectorstore 로드
vectorstore = Chroma(
    persist_directory="./chroma_db",
    collection_name="library",
    embedding_function=OpenAIEmbeddings(model="text-embedding-3-small"),
)

# ═══ 패턴 1 — 전체 컬렉션 검색 ===
results = vectorstore.similarity_search("RAG 성능 평가", k=5)

# ═══ 패턴 2 — 특정 책만 ===
results = vectorstore.similarity_search(
    "RAG 성능 평가",
    k=5,
    filter={"doc_id": "RAG_handbook_2024"}
)
```

⬇️ 코드가 길어 다음 장에 이어집니다.

2.langchain/7.RAG/3.vectorstore/3.chromadb2_saveload_multifil

패턴 (이어서)

```
# === 패턴 3 — 여러 책 중에서 ===
results = vectorstore.similarity_search(
    "RAG 성능 평가",
    k=5,
    filter={"doc_id": {"$in": ["book_a", "book_b", "paper_x"]}}
)

# ═══ 패턴 4 — 답변 + 출처 인용 ===
for d in results:
    print(f"[{d.metadata['doc_id']} p.{d.metadata['page']}]")
    print(d.page_content[:200])
    print("---")

# ═══ 패턴 5 — Retriever 추상화 (RAG chain 에서) ===
retriever = vectorstore.as_retriever(
    search_kwargs={
        "k": 4,
        "filter": {"doc_id": current_book_id},
    }
)
```

PART E

PART E
운영 팁 — 색인 / 중복 / 재색인

> 본선: 결정적 ID 로 중복 방지 · 심화: 배치·재색인 비용·백엔드 내부 — 약 15분

OpenAI Embedding API 의 batch 한도 활용

단일 호출 vs 배치

```
# ❌ 한 chunk 씩 호출 — 매 호출당 latency
for chunk in chunks:                                    # 1000 chunks
    emb = client.embeddings.create(model=..., input=chunk.text)
# → 약 1000 × 200ms = 200초 (네트워크 + 모델 호출)

# ✅ 배치 호출 — 한 번에 100개씩
BATCH = 100
for i in range(0, len(chunks), BATCH):
    batch = chunks[i:i+BATCH]
    emb = client.embeddings.create(
        model="text-embedding-3-small",
        input=[c.text for c in batch],     # 리스트 전달
    )
# → 약 10 호출 × 500ms = 5초
```

LangChain 자동 배치 + 벤치마크

LangChain `Chroma.from_documents` 의 자동 배치

- 내부적으로 OpenAI Embeddings 의 `embed_documents()` 호출
- 기본 batch size = 1000 (OpenAI 한도)
- chunk 수가 매우 많으면 `chunk_size=512` 처럼 더 작게 (안정성)

1만 chunks 색인 벤치마크

방식	시간	비용 (small)
1개씩 호출	~50분	$0.20
배치 100	**~3분**	**$0.20** (동일)
배치 1000	~2분	$0.20

같은 PDF 를 재색인해도 중복 없이 한 번만 반영

```
import hashlib
from langchain_chroma import Chroma

def deterministic_id(chunk) → str:
    """문서 내용 + 출처 + 페이지로 고유 ID 생성."""
    payload = f"{chunk.metadata['doc_id']}|p{chunk.metadata['page']}|{chunk.page_content}"
    return hashlib.sha256(payload.encode()).hexdigest()[:24]

# 색인 시 ids 전달 — ChromaDB 는 같은 ID upsert (덮어쓰기)
vectorstore.add_documents(
    documents=chunks,
    ids=[deterministic_id(c) for c in chunks],
)
```

Idempotent 색인의 운영 효과

1. **같은 문서 재색인 가능** — 중복 없음
2. **부분 수정** — 바뀐 chunk 만 ID 가 달라짐 → 자동 upsert
3. **삭제도 정확** — `collection.delete(ids=[...])` 로 정밀 삭제

변경 감지 — 추가·삭제 자동 동기화

```
# 파일이 바뀌었는지 mtime + hash 로 체크
existing_ids = set(collection.get(where={"doc_id": pdf_name})["ids"])
new_ids = {deterministic_id(c) for c in new_chunks}

to_add    = new_ids - existing_ids
to_remove = existing_ids - new_ids

# 없어진 chunk 삭제
if to_remove:
    collection.delete(ids=list(to_remove))

# 새 chunk 만 추가
add_chunks = [c for c in new_chunks if deterministic_id(c) in to_add]
collection.add(documents=add_chunks, ids=[deterministic_id(c) for c in add_chunks])
```

> 💡 색인 전체를 다시 만들지 않고 **바뀐 chunk 만** 갱신 — 대용량 컬렉션 운영의 핵심.

"text-embedding-3-small" → "-3-large" 교체 절차

재색인이 필요한 이유

- 임베딩 모델이 다르면 **벡터 공간이 완전히 다름**
- 기존 1536d 벡터와 새 3072d 벡터를 같은 공간에 둘 수 없음
- 검색은 동작하나 의미 없는 결과를 반환

재색인 절차 (안전)

```
# 1. 새 collection 생성 (이름 다르게)
new_vs = Chroma(
    persist_directory="./chroma_db",
    collection_name="library_v2",                          # ← 새 이름
    embedding_function=OpenAIEmbeddings(model="text-embedding-3-large"),
)

# 2. 원본 문서로부터 다시 색인 (chunk 텍스트 그대로 재사용)
old_collection = chromadb.PersistentClient("./chroma_db").get_collection("library")
all_docs = old_collection.get(include=["documents", "metadatas"])
new_vs.add_texts(
    texts=all_docs["documents"],
    metadatas=all_docs["metadatas"],
    ids=all_docs["ids"],   # 같은 ID 유지 → citation 호환
)
```

재색인 비용 추정

비용 추정 — 5만 chunks 재색인

- text-embedding-3-small (1536d): $0.02/1M × 5만 chunks × 평균 500 토큰 = **$0.50**
- text-embedding-3-large (3072d): $0.13/1M × 5만 × 500 = **$3.25**

> 💡 **임베딩 모델 변경 = 색인 비용 + 검증 시간**. 모델 선택은 처음에 신중히.

`./chroma_db/` 디렉토리 구성

```
./chroma_db/
├── chroma.sqlite3                          # collection 메타·doc 본문·메타데이터
├── 32a4f1c0-.../                            # collection UUID
│   ├── header.bin                          # HNSW 인덱스 헤더
│   ├── data_level0.bin                     # HNSW 그래프 (level 0)
│   ├── length.bin
│   ├── link_lists.bin                      # 노드 간 엣지
│   └── *.parquet                           # 임베딩 + ID
└── 9b71... /                                # 다른 collection
```

디스크 사용량 — 1만 chunks (1536d)

- 벡터 본체: 1만 × 1536 × 4B = **60MB** (float32)
- 메타데이터·문서 본문: ~30MB
- HNSW 그래프 (M=32): ~10MB
- **합계 약 100MB** = 한 collection

> 💡 디스크 용량이 부족하면 **차원 축소** (Matryoshka `dimensions=512`) 또는 **PQ 압축** (Milvus/Qdrant).

백엔드 직접 확인 (디버그용)

```
# SQLite — 메타데이터
sqlite3 chroma_db/chroma.sqlite3 \
  "SELECT collection_id, collection_name, count(*) FROM embeddings_queue GROUP BY collection_id"

# Parquet — 벡터 (pandas 로)
python -c "
import pandas as pd, pathlib
for p in pathlib.Path('chroma_db').rglob('*.parquet'):
    df = pd.read_parquet(p)
    print(p, df.shape, df.columns.tolist())
"
```

💡 `chroma.sqlite3` 는 메타·문서, collection UUID 폴더의 `*.parquet` 는 임베딩 벡터를 담는다.

PART F

PART F
미니 실습 — 내 문서 색인

> PDF 5개 → 자기만의 Q&A 봇 — 약 30분

실습 과제 (30분) — 준비 + 색인

1단계 — PDF 준비 (5분)

- 직접 다루는 분야 PDF 3~5개 (논문·매뉴얼·교재)
- `./my_library/` 디렉토리에 모아두기

2단계 — 멀티로딩 + 색인 (10분)

- 위 슬라이드 18 (디렉토리 통째 로딩) 코드 그대로
- 결정적 ID + 메타데이터 풍부화 적용
- 색인 시간·비용 측정

```
# 색인 진행 측정
import time
t = time.time()
vectorstore = Chroma.from_documents(...)
print(f"색인 시간: {time.time()-t:.1f}s, chunks: {len(all_chunks)}")
```

실습 과제 (30분) — 검색 + 답변

3단계 — 3가지 검색 패턴 실험 (10분)

```
# 패턴 A — 전체 검색
vectorstore.similarity_search("Q", k=5)

# 패턴 B — 특정 책만
vectorstore.similarity_search("Q", k=5, filter={"doc_id": "book_a"})

# 패턴 C — 페이지 범위
vectorstore.similarity_search("Q", k=5, filter={
    "$and": [{"doc_id": "book_a"}, {"page": {"$gte": 10}}, {"page": {"$lte": 30}}]
})
```

4단계 — RAG 답변 + 출처 인용 (5분)

- 검색 결과로 RAG 체인 답변 생성
- 답변 끝에 `[doc_id p.N]` 형식으로 출처 표시

결과 보고

- 어떤 질문에 어떤 필터가 효과적이었는가?
- 메타데이터 필터 없을 때 vs 있을 때 — 정확도 차이 한 줄

핵심 8가지

1. ✅ FAISS 의 4가지 한계 (메모리·필터·동시쓰기·하이브리드)
2. ✅ **영속화** = InMemory 와의 결정적 차이 — 재실행 시 재임베딩 비용 0
3. ✅ ChromaDB 저장 구조 — 컬렉션 {id·embedding·document·metadata} + HNSW + persist
4. ✅ Collection CRUD — `add / query / get / update / delete`
5. ✅ 임베딩 함수 3가지 (default SBERT / OpenAI / multilingual-e5)
6. ✅ 메타데이터 필터 — `$eq · $gt · $in · $and · where_document.$contains`
7. ✅ **결정적 ID** = 중복 방지 + 부분 수정 = idempotent 색인
8. ✅ (심화) 6종 비교 · 백엔드(SQLite·Parquet) · 재색인 비용

실습 과제 — 직접 해보기

필수

- [ ] PDF 5개 → ChromaDB 컬렉션 구축
- [ ] 메타데이터 필터 3가지 패턴 실험 (전체 / 특정 책 / 페이지 범위)
- [ ] 색인 시간·비용 기록

옵션

- [ ] **multilingual-e5-large** 로 같은 데이터 재색인 → OpenAI 와 검색 결과 비교
- [ ] `2.langchain/7.RAG/8.web_app/` 실행 — Flask UI 챗봇과 결합
- [ ] **5만 chunks** 이상 — 색인 시간 + 디스크 측정

📤 정리

- 색인 결과 (chunks 수, 디스크 용량, 색인 시간) — 한 줄