Retrieval Service 문서

목차

개요

Retrieval Service는 Yonsei Research Assistant의 핵심 검색 엔진으로, 다중 데이터 소스에서 학술 자료를 검색하고, 재순위화(reranking)하며, 품질을 평가(CRAG)하는 통합 검색 파이프라인을 제공합니다.

주요 기능

- **다중 소스 통합 검색**: 연세대학교 도서관 소장자료, 전자자료, 국립중앙도서관 벡터 DB
- **병렬 처리**: 여러 데이터 소스에서 동시 검색으로 성능 최적화
- **Reranking & Fusion**: Cross-encoder 기반 문서 재순위화 및 다양한 융합 전략
- **CRAG (Corrective RAG)**: LLM 기반 문서 품질 평가 및 필터링
- **웹 검색 필요성 판단**: 검색 결과 품질이 낮을 경우 자동 감지

서비스 정보

- **포트**: 8003
- **엔드포인트**:
  - `POST /search`: 메인 검색 API
  - `GET /health`: 서비스 및 데이터 소스 상태 확인
  - `GET /`: 서비스 정보

서비스 구조

```
retrieval_service/
├── main.py                          # FastAPI 애플리케이션 진입점
├── config.py                        # 서비스 설정 (환경 변수, 모델 설정)
│
├── services/                        # 핵심 비즈니스 로직
│   ├── search_executor.py          # 🔥 전체 검색 파이프라인 조정자
│   ├── retriever.py                # 다중 소스 검색 실행
│   ├── ranker.py                   # Reranking & Fusion
│   └── refiner.py                  # CRAG 품질 평가 및 필터링
│
```

```
├── adapters/                          # 데이터 소스별 어댑터
│   ├── base_adapters.py               # BaseRetriever 인터페이스
│   ├── library_holdings_adapter.py    # 연세대 소장자료 어댑터
│   ├── electronic_resources_adapter.py # 연세대 전자자료 어댑터
│   └── vectordb_adapter.py            # FAISS 벡터 DB 어댑터
│
├── scrapers/                          # 웹 스크래핑 엔진
│   ├── base_scraper.py                # 공통 스크래핑 로직 (로그인, 세션 관리)
│   ├── library_holdings_scraper.py    # 소장자료 스크래퍼
│   ├── electronic_resources_scraper.py # 전자자료 스크래퍼
│   └── search_params.py               # 검색 파라미터 모델
│
├── data/                              # 벡터 DB 데이터
│   ├── faiss.index                    # FAISS 인덱스
│   └── build_faiss_index.py           # 인덱스 빌드 스크립트
│
└── tests/
    └── test_step1_retrieval.py        # 검색 기능 테스트
```

핵심 검색 파이프라인 (SearchExecutor)

`SearchExecutor`는 전체 검색 프로세스를 조율하는 핵심 클래스입니다. **5단계 파이프라인**으로 구성됩니다.

파이프라인 개요

```python
class SearchExecutor:
    def __init__(self):
        self.retriever = RetrieverService()  # Step 1: 검색
        self.ranker = RankerService()        # Step 2: Reranking
        self.refiner = RefinerService()      # Step 3-5: CRAG 평가 및 필터링
```

🔥 5단계 검색 프로세스

Step 1: 다중 소스 검색 (Retriever)

```python
# 모든 데이터 소스에서 병렬 검색
raw_documents = await self.retriever.retrieve_all(request)
```

역할:

- Strategy Service로부터 받은 `SearchRequest`를 각 어댑터에 전달
- 여러 데이터 소스(연세대 소장자료, 전자자료, 벡터 DB)에서 **병렬 검색**
- 각 어댑터는 `request_to_search_params()`로 요청을 변환 후 `search()` 실행

주요 특징:

- `asyncio.gather()`를 사용한 비동기 병렬 처리
- Vector DB는 deadlock 방지를 위해 별도 처리
- 검색 실패 시 해당 소스만 스킵하고 다른 소스는 계속 진행

출력:

- `List[Document]`: 중복 제거되지 않은 원본 문서 리스트

Step 2: Reranking & Fusion (Ranker)

```
# Cross-encoder로 재순위화 및 융합
ranked_documents = self.ranker.rerank_and_fuse(
    documents=raw_documents,
    user_query=request.user_query
)
```

역할:

- **중복 제거**: Content 기반 해싱으로 동일 문서 제거
- **Cross-encoder Reranking**: 사용자 쿼리와 각 문서의 관련성을 깊이 있게 재평가
- **Fusion 전략 적용**: 여러 소스의 결과를 통합

Fusion 전략 옵션:

1. **RRF (Reciprocal Rank Fusion)**:

   ```
   RRF_score = Σ 1/(k + rank_i)  # k=60 (기본값)
   ```

   - 여러 소스의 순위를 조화롭게 융합
   - 어느 한 소스에 편향되지 않음

2. **Weighted Fusion**:

   ```
   weighted_score = cross_encoder_score × source_weight
   ```

   - 소스별 가중치 적용 (예: vector_db=0.6, library=0.4)

3. **Cross-encoder Only**:

   - Cross-encoder 점수만 사용

현재 상태:

- ⚠ `tmp_rerank_and_fuse()` 사용 중 (무작위 정렬)
- TODO: Rerank 모델 파인튜닝 완료 후 `rerank_and_fuse()`로 전환

출력:

- `List[RankedDocument]`: 재순위화된 문서 (top_k개, 기본 20개)

Step 3: CRAG 품질 평가 (Refiner)

```
# LLM으로 각 문서의 관련성 평가
crag_results = await self.refiner.evaluate_relevance(
    documents=ranked_documents,
    user_query=request.user_query
)
```

역할:

- **CRAG (Corrective RAG)**: LLM(Gemini 1.5 Flash)을 사용해 각 문서를 평가
- 사용자 질문과 문서 내용의 관련성을 3단계로 분류

관련성 등급:

```
class RelevanceLevel(Enum):
    CORRECT = "correct"      # 바로 사용 가능 - 질문에 직접 답변
    AMBIGUOUS = "ambiguous"  # 부분적 관련 - 추가 정보 필요
    INCORRECT = "incorrect"  # 무관 또는 오해의 소지
```

평가 프롬프트:

```
질문: {user_query}
문서 내용: {document_content}

다음 중 하나로 판단:
- CORRECT: 직접적으로 답변, 관련성 높음
- AMBIGUOUS: 부분적 관련, 추가 정보 필요
- INCORRECT: 무관하거나 오해 소지

JSON 응답:
{
  "relevance": "CORRECT" | "AMBIGUOUS" | "INCORRECT",
  "confidence": 0.0~1.0,
  "reason": "판단 근거"
}
```

출력:

- `List[CRAGResult]`: 각 문서의 평가 결과 (relevance, confidence, reason)

Step 4: 품질 필터링 (Refiner)

```
# CRAG 결과 기반 문서 필터링
filtered_documents = self.refiner.filter_by_quality(crag_results)
```

역할:

- CRAG 평가를 기반으로 문서 필터링

필터링 규칙:

```
if relevance == CORRECT:
    ✅ 포함  # 그대로 사용
elif relevance == AMBIGUOUS:
    if confidence >= CRAG_RELEVANCE_THRESHOLD:  # 기본 0.6
        ✅ 포함  # 신뢰도 충분
    else:
        ❌ 제외  # 신뢰도 낮음
elif relevance == INCORRECT:
    ❌ 제외  # 무조건 제외
```

출력:

- `List[RankedDocument]`: 품질 검증된 최종 문서

Step 5: 웹 검색 필요성 판단 (Refiner)

```
# INCORRECT 비율이 높으면 웹 검색 필요
needs_web = self.refiner.needs_web_search(crag_results)
```

역할:

- 내부 검색 결과의 품질이 낮을 경우 외부 웹 검색 필요 여부 판단

판단 기준:

```
incorrect_ratio = incorrect_count / total_documents

if incorrect_ratio > CRAG_INCORRECT_RATIO_THRESHOLD:  # 기본 0.7
    needs_web_search = True  # 70% 이상이 INCORRECT면 웹 검색
```

출력:

- `bool`: 웹 검색 필요 여부

최종 결과 반환

```
return RetrievalResult(
    documents=filtered_documents,          # 최종 문서 리스트
    crag_analysis=crag_results,           # 전체 CRAG 평가 결과
    metadata={                             # 검색 통계
        'processing_time_seconds': elapsed_time,
        'total_retrieved': len(raw_documents),
        'after_rerank': len(ranked_documents),
        'after_crag': len(filtered_documents),
        'sources_used': [...]
    },
    needs_web_search=needs_web             # 웹 검색 필요 플래그
)
```

주요 컴포넌트

1. RetrieverService

위치: `services/retriever.py`

역할:

- Strategy Service의 라우팅 결정을 실행
- 여러 데이터 소스에서 병렬 검색 수행

주요 메서드:

```
async def retrieve_all(self, request: SearchRequest) -> List[Document]:
    """
    모든 데이터 소스에서 병렬 검색

    Process:
    1. request.routes에서 지정된 소스별 어댑터 선택
    2. 각 어댑터의 request_to_search_params() 호출
    3. asyncio.gather()로 병렬 검색 실행
    4. Vector DB는 별도 처리 (deadlock 방지)
    5. 모든 결과 병합 후 반환
    """
```

어댑터 등록:

```
self.adapters: Dict[RetrievalRoute, BaseRetriever] = {
    RetrievalRoute.YONSEI_HOLDINGS: LibraryHoldingsAdapter(),
    RetrievalRoute.YONSEI_ELECTRONICS: ElectronicResourcesAdapter(),
```

```
    RetrievalRoute.VECTOR_DB: VectorDBAdapter()
}
```

병렬 처리 로직:

```
# Vector DB 제외한 다른 소스들 병렬 실행
tasks = []
for route in request.routes:
    if route is not RetrievalRoute.VECTOR_DB:
        tasks.append(self._retrieve_process(adapter, request, route))

results = await asyncio.gather(*tasks, return_exceptions=True)

# Vector DB는 별도로 순차 처리
if RetrievalRoute.VECTOR_DB in request.routes:
    vector_docs = await vector_adapter.search(params, top_k)
    results.append(vector_docs)
```

---

2. RankerService

위치: `services/ranker.py`

역할:

- 여러 소스의 검색 결과를 융합하고 재순위화
- Cross-encoder 모델로 쿼리-문서 관련성 재평가

주요 메서드:

```
def rerank_and_fuse(
    self,
    documents: List[Document],
    user_query: str,
    method: str = "rrf"
) -> List[RankedDocument]:
    """
    Rerank + Fusion 파이프라인

    Steps:
    1. _deduplicate(): Content 기반 중복 제거
    2. _cross_encoder_rerank(): Cross-encoder로 재점수
    3. Fusion 전략 적용 (RRF/Weighted/Cross-encoder)
    4. Top-K 필터링 및 순위 부여
    """
```

Fusion 전략들:

1. **RRF (Reciprocal Rank Fusion)**:

```
def _reciprocal_rank_fusion(self, documents, k=60):
    """
    여러 검색 결과의 순위를 조화롭게 융합

    Formula: RRF_score = Σ 1/(k + rank_i)

    Example:
    - Source A: rank 1 → 1/(60+1) = 0.0164
    - Source B: rank 5 → 1/(60+5) = 0.0154
    - Final: 0.0164 + 0.0154 = 0.0318
    """
```

2. **Weighted Fusion**:

```
def _weighted_fusion(self, documents, weights=None):
    """
    소스별 신뢰도 가중치 적용

    Default weights:
    - vector_db: 0.6
    - yonsei_library: 0.4
    """
```

현재 사용 중인 임시 메서드:

```
def tmp_rerank_and_fuse(self, documents, user_query):
    """
    임시 Rerank + Fusion (단순 무작위 정렬)
    TODO: Rerank 모델 파인튜닝 후 제거
    """
```

---

3. RefinerService

위치: `services/refiner.py`

역할:

- CRAG (Corrective RAG) - 검색 결과 품질 평가
- 관련성 기반 필터링 및 웹 검색 필요성 판단

주요 메서드:

```
async def evaluate_relevance(
    self,
    documents: List[RankedDocument],
    user_query: str
) -> List[CRAGResult]:
    """
    각 문서의 관련성을 CORRECT/AMBIGUOUS/INCORRECT로 평가

    Process:
    1. 각 문서에 대해 _evaluate_single_document() 호출
    2. LLM(Gemini 1.5 Flash)으로 관련성 평가
    3. JSON 응답 파싱 → CRAGResult 생성
    4. 실패 시 AMBIGUOUS로 폴백
    """
```

```
def filter_by_quality(
    self,
    crag_results: List[CRAGResult]
) -> List[RankedDocument]:
    """
    CRAG 평가 기반 문서 필터링

    Rules:
    - CORRECT: 그대로 사용
    - AMBIGUOUS: confidence >= threshold (0.6) 이면 포함
    - INCORRECT: 제거
    """
```

```
def needs_web_search(self, crag_results: List[CRAGResult]) -> bool:
    """
    INCORRECT 비율이 70% 이상이면 외부 웹 검색 필요

    Formula:
    incorrect_ratio = incorrect_count / total_documents
    return incorrect_ratio > 0.7
    """
```

LLM 설정:

```
def __init__(self):
    genai.configure(api_key=settings.GEMINI_API_KEY)
    self.model = genai.GenerativeModel(
        model_name="gemini-1.5-flash",
        generation_config={"response_mime_type": "application/json"}
    )
```

4. 어댑터 (Adapters)

모든 어댑터는 `BaseRetriever` 인터페이스를 구현합니다.

BaseRetriever 인터페이스

위치: `adapters/base_adapters.py`

```python
class BaseRetriever(ABC):
    """모든 검색 어댑터가 구현해야 하는 인터페이스"""

    @abstractmethod
    async def request_to_search_params(self, request: SearchRequest):
        """SearchRequest를 어댑터별 검색 파라미터로 변환"""
        pass

    @abstractmethod
    async def search(self, search_params, top_k: int) -> List[Document]:
        """통일된 검색 메서드"""
        pass

    @abstractmethod
    async def health_check(self) -> bool:
        """데이터 소스 연결 상태 확인"""
        pass

    @property
    @abstractmethod
    def source_name(self) -> str:
        """데이터 소스 식별자"""
        pass
```

LibraryHoldingsAdapter

위치: `adapters/library_holdings_adapter.py`

역할: 연세대학교 도서관 소장자료(단행본, 학위논문 등) 검색

주요 기능:

- `SearchRequest` → `LibraryHoldingsSearchParams` 변환
- `LibraryHoldingsScraper`를 사용한 실제 스크래핑
- `LibraryHoldingInfo` → `Document` 표준화

예시:

```python
# SearchRequest 변환
search_params = LibraryHoldingsSearchParams(
```

```
    query="인공지능",
    search_field=LibrarySearchField.TITLE,
    additional_queries=[
        {
            "search_field": LibrarySearchField.AUTHOR,
            "query": "이중원",
            "operator": QueryOperator.AND
        }
    ],
    year_range={"from_year": 2020, "to_year": 2024},
    material_types=[HoldingsMaterialType.BOOK]
)

# 검색 실행
documents = await adapter.search(search_params, top_k=10)
```

문서 표준화:

```
# LibraryHoldingInfo → Document
Document(
    content=f"{holding.title}\n\n{holding.book_description}",
    metadata={
        "title": holding.title,
        "author": holding.author,
        "publication_year": holding.publication_year,
        "isbn": holding.isbn,
        "detail_url": holding.detail_url,
        "data_source": "yonsei_holdings"
    },
    score=1.0 / (rank + 1)  # 순위 기반 스코어
)
```

ElectronicResourcesAdapter

위치: `adapters/electronic_resources_adapter.py`

역할: 연세대학교 도서관 전자자료(학술논문, E-Book, 저널 등) 검색

주요 기능:

- `SearchRequest` → `ElectronicResourcesSearchParams` 변환
- `ElectronicResourcesScraper`를 사용한 스크래핑
- `ElectronicResourceInfo` → `Document` 표준화

필터 지원:

- `year_range`: 출판 연도 범위
- `academic_journals_only`: 학술지 논문만
- `foreign_language`: 외국어 자료 포함 여부

문서 표준화 예시:

```
Document(
    content=f"{resource.title}\n\n{resource.abstract}",
    metadata={
        "title": resource.title,
        "author": resource.author,
        "publication_year": resource.publication_year,
        "doi": resource.doi,
        "link_url": resource.link_url,
        "keywords": resource.keywords,
        "data_source": "yonsei_electronics"
    },
    score=1.0 / (rank + 1)
)
```

VectorDBAdapter

위치: `adapters/vectordb_adapter.py`

역할: FAISS 기반 국립중앙도서관 도서 벡터 DB 검색

주요 기능:

- 쿼리 임베딩 생성 (KURE-v1 모델)
- FAISS 인덱스 검색
- 메타데이터 조회 (Pickle + SQLite)

쿼리 처리 로직:

```
# AND 연산자: 쿼리 결합
if operator_1 == QueryOperator.AND:
    query_1 += " " + query_2

# OR 연산자: 별도 벡터 생성
if operator_1 == QueryOperator.OR:
    vector_1 = encode(query_1)
    vector_2 = encode(query_2)
    # 각각 검색 후 결과 병합

# NOT 연산자: 무시 (벡터 검색에서 지원 어려움)
```

검색 프로세스:

```
# 1. 쿼리 임베딩
encoder = SentenceTransformer("nlpai-lab/KURE-v1")
vector = encoder.encode([query])
```

```
# 2. FAISS 검색
distances, indices = index.search(vector, top_k)

# 3. 메타데이터 조회
metadata_id = metadata_faiss_map[faiss_id]
metadata = sqlite_db.query(metadata_id)

# 4. Document 생성
Document(
    content=metadata['title'] + "\n" + metadata['description'],
    metadata={
        "title": metadata['title'],
        "author": metadata['author'],
        "isbn": metadata['isbn'],
        "data_source": "vector_book_db"
    },
    score=1.0 / (1.0 + distance)
)
```

데이터 구조:

- `faiss.index`: FAISS IVF 인덱스 파일
- `id_to_metadata.pkl`: FAISS ID → 메타데이터 ID 매핑
- `metadata.db`: SQLite 메타데이터 DB

---

5. 스크래퍼 (Scrapers)

모든 스크래퍼는 `BaseLibraryScraper`를 상속합니다.

BaseLibraryScraper

위치: `scrapers/base_scraper.py`

역할:

- 연세대학교 도서관 로그인 처리
- 세션 관리 (aiohttp + Playwright)
- 공통 스크래핑 유틸리티

주요 기능:

```
async def perform_login(self, user_id: str, user_pw: str) -> bool:
    """
    Playwright로 브라우저 자동화 로그인

    Process:
    1. Playwright로 headless 브라우저 실행
    2. 로그인 페이지 이동
    3. 아이디/비밀번호 입력
```

```
    4. 로그인 버튼 클릭
    5. 생성된 쿠키를 aiohttp 세션으로 복사
    """
```

로그인이 필요한 이유:

- 전자자료 검색은 로그인 필요
- 소장자료 검색은 로그인 불필요 (하지만 상세 정보 조회 시 유리)

세션 관리:

```
# aiohttp 세션 생성
self.session = aiohttp.ClientSession(headers=self.headers)

# Playwright 쿠키 → aiohttp 세션
for cookie in playwright_cookies:
    self.session.cookie_jar.update_cookies({
        cookie['name']: cookie['value']
    })
```

LibraryHoldingsScraper

위치: `scrapers/library_holdings_scraper.py`

역할: 연세대학교 도서관 소장자료 스크래핑

주요 메서드:

```
async def search(self, params: LibraryHoldingsSearchParams) ->
List[LibraryHoldingInfo]:
    """
    소장자료 검색

    Process:
    1. 검색 파라미터를 URL 쿼리 스트링으로 변환
    2. 검색 결과 페이지 요청 (BeautifulSoup 파싱)
    3. 각 결과 항목에서 기본 정보 추출
    4. 상세 페이지 요청 (병렬 처리)
    5. LibraryHoldingInfo 객체 생성
    """
```

스크래핑 대상 필드:

- 제목 (title)
- 저자 (author)
- 자료 유형 (material_type)
- 발행 사항 (publication_info)

- ISBN
- 책 소개 (book_description)
- 상세 URL (detail_url)

ElectronicResourcesScraper

위치: `scrapers/electronic_resources_scraper.py`

역할: 연세대학교 도서관 전자자료 스크래핑

주요 메서드:

```
async def search(self, params: ElectronicResourcesSearchParams) ->
List[ElectronicResourceInfo]:
    """
    전자자료 검색

    Process:
    1. 로그인 (필수)
    2. 검색 파라미터를 API 요청으로 변환
    3. 검색 결과 페이지 파싱
    4. 각 자료의 초록, DOI, 원문 링크 추출
    5. ElectronicResourceInfo 객체 생성
    """
```

스크래핑 대상 필드:

- 제목 (title)
- 저자 (author)
- 출처 (source): 저널명, 권호, 페이지
- 출판년 (publication_year)
- DOI
- 원문 링크 (link_url)
- 초록 (abstract)
- 키워드 (keywords)

데이터 모델

요청 모델

SearchRequest

Strategy Service → Retrieval Service 요청 모델

```
class SearchRequest(BaseModel):
    queries: SearchQueries              # 검색 쿼리 (최대 3개)
    routes: List[RetrievalRoute]        # 검색 소스들
```

```
    filters: Optional[Dict[str, Any]]  # 필터 조건
    top_k: int = 10                    # 각 소스별 반환 문서 수
    user_query: str                    # 원본 사용자 질문 (CRAG용)
```

예시:

```
SearchRequest(
    queries=SearchQueries(
        query_1="artificial intelligence",
        search_field_1=ElectronicSearchField.TITLE,
        operator_1=QueryOperator.AND,
        query_2="machine learning",
        search_field_2=ElectronicSearchField.TITLE
    ),
    routes=[
        RetrievalRoute.YONSEI_ELECTRONICS,
        RetrievalRoute.VECTOR_DB
    ],
    filters={
        "year_range": (2020, 2024),
        "academic_journals_only": True
    },
    top_k=10,
    user_query="AI와 머신러닝에 관한 최근 학술논문"
)
```

SearchQueries

멀티 쿼리 모델 (최대 3개 쿼리)

```
class SearchQueries(BaseModel):
    query_1: str                          # 필수
    search_field_1: Union[LibrarySearchField, ElectronicSearchField]
    operator_1: Optional[QueryOperator] # AND/OR/NOT

    query_2: Optional[str]
    search_field_2: Optional[Union[LibrarySearchField,
ElectronicSearchField]]
    operator_2: Optional[QueryOperator]

    query_3: Optional[str]
    search_field_3: Optional[Union[LibrarySearchField,
ElectronicSearchField]]
```

검증 규칙:

- 쿼리는 순차적으로만 입력 가능: (query_1), (query_1, query_2), (query_1, query_2, query_3)

- query_2가 있으면 search_field_2 필수
- query_3이 있으면 operator_2, search_field_3 필수

응답 모델

RetrievalResult

Retrieval Service의 최종 응답

```
class RetrievalResult(BaseModel):
    documents: List[RankedDocument]    # CRAG 필터링 + Rerank 완료 문서
    crag_analysis: List[CRAGResult]    # 전체 문서 CRAG 평가
    metadata: Dict[str, Any]           # 검색 통계
    needs_web_search: bool             # 웹 검색 필요 여부
```

메타데이터 예시:

```
{
    'processing_time_seconds': 2.34,
    'total_retrieved': 45,
    'after_rerank': 20,
    'after_crag': 12,
    'sources_used': ['yonsei_electronics', 'vector_book_db']
}
```

Document

검색된 원본 문서

```
class Document(BaseModel):
    content: str                        # 문서 본문 텍스트
    metadata: Dict[str, Any]           # 출처, 제목, 저자, URL 등
    score: float = 0.0                 # 검색 유사도 점수
    doc_id: Optional[str] = None       # 문서 고유 ID
```

RankedDocument

Rerank 후 최종 문서

```
class RankedDocument(BaseModel):
    content: str
    metadata: Dict[str, Any]
```

```
    rerank_score: float                # Cross-encoder 재점수
    original_score: float              # 초기 검색 점수
    source: str                        # 데이터 소스
    rank: int                          # 최종 순위 (1부터 시작)
```

CRAGResult

CRAG 품질 평가 결과

```
class CRAGResult(BaseModel):
    document: RankedDocument
    relevance: RelevanceLevel          # CORRECT/AMBIGUOUS/INCORRECT
    confidence: float                  # 0.0~1.0
    reason: Optional[str]              # 판단 근거
```

Enum 타입

RetrievalRoute

```
class RetrievalRoute(str, Enum):
    VECTOR_DB = "vector_book_db"               # 국립중앙도서관 도서 벡터 DB
    YONSEI_HOLDINGS = "yonsei_holdings"        # 연세대 도서관 소장자료
    YONSEI_ELECTRONICS = "yonsei_electronics" # 연세대 도서관 전자자료
```

QueryOperator

```
class QueryOperator(str, Enum):
    AND = "and"  # 필수
    OR = "or"    # 선택
    NOT = "not"  # 제외
```

LibrarySearchField

```
class LibrarySearchField(str, Enum):
    TOTAL = "TOTAL"      # 전체
    TITLE = "1"          # 서명(책제목)
    AUTHOR = "2"         # 저자
    PUBLISHER = "3"      # 출판사
    SUBJECT = "4"        # 주제어
```

ElectronicSearchField

```
class ElectronicSearchField(str, Enum):
    TOTAL = ""       # 전체
    KEYWORD = "TX"   # 키워드
    TITLE = "TI"     # 제목
    AUTHOR = "AU"    # 저자
    SUBJECT = "SU"   # 주제어
```

사용 예시

1. 기본 검색

```
from shared.models import (
    SearchRequest,
    SearchQueries,
    RetrievalRoute,
    LibrarySearchField
)
from retrieval_service.services.search_executor import SearchExecutor

# 검색 요청 생성
request = SearchRequest(
    queries=SearchQueries(
        query_1="인공지능",
        search_field_1=LibrarySearchField.TITLE
    ),
    routes=[RetrievalRoute.YONSEI_HOLDINGS],
    filters={
        "year_range": (2020, 2024),
        "material_types": ["BOOK"]
    },
    top_k=10,
    user_query="인공지능에 관한 최근 도서"
)

# 검색 실행
executor = SearchExecutor()
result = await executor.execute(request)

# 결과 출력
print(f"총 {len(result.documents)}개 문서 검색")
for doc in result.documents[:3]:
    print(f"제목: {doc.metadata['title']}")
    print(f"순위: {doc.rank}, 점수: {doc.rerank_score}")
```

---

2. 다중 쿼리 검색

```
request = SearchRequest(
    queries=SearchQueries(
        query_1="artificial intelligence",
        search_field_1=ElectronicSearchField.TITLE,
        operator_1=QueryOperator.AND,
        query_2="machine learning",
        search_field_2=ElectronicSearchField.TITLE,
        operator_2=QueryOperator.OR,
        query_3="deep learning",
        search_field_3=ElectronicSearchField.TITLE
    ),
    routes=[RetrievalRoute.YONSEI_ELECTRONICS],
    filters={
        "year_range": (2020, 2024),
        "academic_journals_only": True
    },
    top_k=10,
    user_query="AI와 머신러닝, 딥러닝 논문"
)
```

쿼리 해석:

- (AI AND ML) OR 딥러닝
- 제목에 해당 키워드 포함
- 2020-2024년 학술논문만

---

3. 다중 소스 검색

```
request = SearchRequest(
    queries=SearchQueries(
        query_1="자연어처리",
        search_field_1=LibrarySearchField.SUBJECT
    ),
    routes=[
        RetrievalRoute.YONSEI_HOLDINGS,       # 소장자료
        RetrievalRoute.YONSEI_ELECTRONICS,    # 전자자료
        RetrievalRoute.VECTOR_DB              # 벡터 DB
    ],
    top_k=5,  # 각 소스당 5개씩
    user_query="자연어처리 관련 자료"
)

result = await executor.execute(request)

# 소스별 문서 수 확인
print(result.metadata['sources_used'])  # ['yonsei_holdings',
'yonsei_electronics', 'vector_book_db']
print(f"총 검색: {result.metadata['total_retrieved']}")  # 약 15개
print(f"최종 반환: {len(result.documents)}")  # CRAG 필터링 후
```

4. CRAG 분석 결과 확인

```
result = await executor.execute(request)

# CRAG 통계
correct = sum(1 for r in result.crag_analysis if r.relevance == "correct")
ambiguous = sum(1 for r in result.crag_analysis if r.relevance ==
"ambiguous")
incorrect = sum(1 for r in result.crag_analysis if r.relevance ==
"incorrect")

print(f"CORRECT: {correct}, AMBIGUOUS: {ambiguous}, INCORRECT:
{incorrect}")

# 각 문서의 평가 결과
for crag in result.crag_analysis[:3]:
    print(f"\n제목: {crag.document.metadata['title']}")
    print(f"관련성: {crag.relevance}")
    print(f"신뢰도: {crag.confidence:.2f}")
    print(f"이유: {crag.reason}")

# 웹 검색 필요 여부
if result.needs_web_search:
    print("⚠ 검색 결과 품질이 낮습니다. 웹 검색을 권장합니다.")
```

5. API 엔드포인트 호출

```
import requests

# FastAPI 서버 실행 (localhost:8003)
# POST /search
response = requests.post(
    "http://localhost:8003/search",
    json={
        "queries": {
            "query_1": "blockchain",
            "search_field_1": "TITLE"
        },
        "routes": ["yonsei_holdings"],
        "top_k": 10,
        "user_query": "블록체인 관련 도서"
    }
)

result = response.json()
print(f"검색 완료: {len(result['documents'])}개 문서")
```

6. Health Check

```
# GET /health
response = requests.get("http://localhost:8003/health")
status = response.json()

print(f"서비스 상태: {status['status']}")  # healthy / degraded
print("어댑터 상태:")
for adapter, is_healthy in status['adapters'].items():
    print(f"  - {adapter}: {'✅' if is_healthy else '❌'}")
```

설정 및 환경 변수

config.py

위치: `retrieval_service/config.py`

```
class Settings(BaseSettings):
    # 연세대학교 계정 (스크래핑용)
    YONSEI_ID: str
    YONSEI_PW: str

    # 서비스 기본 정보
    SERVICE_NAME: str = "retrieval-service"
    SERVICE_PORT: int = 8003

    # VectorDB (FAISS + SQLite) 설정
    FAISS_INDEX_PATH: str
    FAISS_ID_TO_METADATA_PATH: str
    METADATA_DB_PATH: str

    # 임베딩 모델
    VECTOR_EMBEDDING_MODEL: str = "nlpai-lab/KURE-v1"
    VECTOR_DIMENSION: int = 1024

    # Reranking 설정
    RERANK_MODEL: str = "BAAI/bge-reranker-v2-m3"
    RERANK_TOP_K: int = 20
    FUSION_METHOD: str = "rrf"  # 'rrf' | 'weighted' | 'cross_encoder'

    # CRAG 설정
    CRAG_LLM_MODEL: str = "gemini-1.5-flash"
    CRAG_RELEVANCE_THRESHOLD: float = 0.6
    CRAG_INCORRECT_RATIO_THRESHOLD: float = 0.7

    # API 키
    GEMINI_API_KEY: str
```

```
    # 로깅
    LOG_LEVEL: str = "INFO"
```

.env 파일

```
# 연세대학교 계정
YONSEI_ID=your_student_id
YONSEI_PW=your_password

# VectorDB 경로
FAISS_INDEX_PATH=/path/to/faiss.index
FAISS_ID_TO_METADATA_PATH=/path/to/id_to_metadata.pkl
METADATA_DB_PATH=/path/to/metadata.db

# Gemini API Key
GEMINI_API_KEY=your_gemini_api_key
```

주요 특징 정리

1. **다중 소스 통합**

- 연세대 도서관 소장자료
- 연세대 도서관 전자자료
- 국립중앙도서관 벡터 DB
- 통일된 `Document` 모델로 표준화

2. **병렬 처리**

- `asyncio.gather()`로 여러 소스 동시 검색
- Vector DB는 deadlock 방지를 위해 별도 처리
- 성능 최적화

3. **고급 Reranking**

- Cross-encoder 모델 (BAAI/bge-reranker-v2-m3)
- RRF, Weighted, Cross-encoder Fusion 지원
- 소스별 가중치 조정 가능

4. **CRAG 품질 평가**

- LLM(Gemini 1.5 Flash) 기반 관련성 평가
- CORRECT/AMBIGUOUS/INCORRECT 3단계 분류
- 신뢰도 기반 필터링

5. **웹 검색 트리거**

- INCORRECT 비율이 70% 이상이면 웹 검색 필요 신호
- 내부 검색 품질 자동 모니터링

6. **유연한 쿼리 구성**

- 최대 3개 쿼리 조합
- AND/OR/NOT 연산자 지원
- 소스별 검색 필드 커스터마이징

7. **상세한 메타데이터**

- 처리 시간, 소스별 문서 수 등 통계
- CRAG 분석 결과 포함
- 디버깅 및 모니터링 용이

제한 사항 및 TODO

현재 제한 사항

1. **Reranking 미완성**
    - 현재: `tmp_rerank_and_fuse()` 사용 (무작위 정렬)
    - TODO: Rerank 모델 파인튜닝 후 `rerank_and_fuse()`로 전환
2. **Vector DB NOT 연산자 미지원**
    - Vector 검색 특성상 NOT 연산자 구현 어려움
    - 현재: NOT 연산자는 무시됨
3. **중복 제거 알고리즘**
    - 현재: Content 해싱 (단순)
    - TODO: Embedding 유사도 기반 중복 제거
4. **로그인 쿠키 관리**
    - 세션 만료 시 재로그인 로직 필요
    - 현재: 수동으로 재시작 필요

향후 개선 사항

- [ ] Rerank 모델 파인튜닝
- [ ] Embedding 기반 중복 제거
- [ ] CRAG 프롬프트 최적화
- [ ] 캐싱 메커니즘 추가
- [ ] 검색 로그 분석 대시보드
- [ ] 웹 검색 통합 (Tavily, Serper 등)

테스트

테스트 스크립트: `tests/test_step1_retrieval.py`

```
# 전자자료 검색 테스트
cd backend
python3 -m retrieval_service.tests.test_step1_retrieval --test electronic

# 소장자료 검색 테스트
python3 -m retrieval_service.tests.test_step1_retrieval --test holdings

# 벡터 DB 검색 테스트
python3 -m retrieval_service.tests.test_step1_retrieval --test vector

# 전체 파이프라인 테스트
python3 -m retrieval_service.tests.test_step1_retrieval --test all
```

참고 자료

- **CRAG 논문**: Corrective Retrieval Augmented Generation
- **RRF 논문**: Reciprocal Rank Fusion
- **Cross-encoder 모델**: BAAI/bge-reranker-v2-m3
- **임베딩 모델**: nlpai-lab/KURE-v1 (한국어 특화)
- **LLM**: Gemini 1.5 Flash (Google)

문서 작성일: 2025-11-22
버전: 1.0.0