Chapter 4-4 복습 자료: 개인화 필터링 시스템 심층 분석

Prologue: 우리 시스템의 새로운 '두뇌'는 어떻게 생각하는가?

우리는 방금 `PersonalizedJobFilter`라는 강력한 두뇌를 우리 시스템에 이식했습니다. 이 두뇌는 단순한 키워드 매칭을 넘어, 신호용 님의 이력서와 커리어 목표를 이해하고, 각 채용 공고에 대해 "이것이 정말 당신에게 가치 있는 기회인가?"를 점수로 판단해줍니다.

이번 복습 자료에서는 이 두뇌의 세 가지 핵심 구성 요소 - **설계도(`YAML`)**, **엔진(`Filter Class`)**, **관제탑(`main.py`)** - 가 어떻게 서로 유기적으로 협력하여 이 놀라운 판단을 내리는지, 그 내부 동작 원리를 집중적으로 탐구합니다.

1. 설계도: `config/job_filter_config.yaml` - 지식과 전략의 보고

이 파일은 우리 필터의 모든 '지식'과 '전략'이 담긴 두뇌의 핵심 메모리입니다.

- **What (무엇인가?):** YAML은 사람이 읽고 쓰기 매우 쉬운 데이터 형식입니다. 파이썬의 딕셔너리나 리스트처럼, 계층적인 구조를 들여쓰기로 표현합니다.
- **Why (왜 코드에서 분리했는가?):**
  - **유연성:** "나 이제 `FastAPI`보다 `NestJS`에 더 집중할래!" 라고 마음이 바뀌었을 때, 복잡한 파이썬 코드를 건드릴 필요 없이, 이 파일에 `nestjs` 키워드만 추가하면 필터의 판단 기준이 즉시 업데이트됩니다.
  - **가독성:** 신호용 님의 기술 스택과 가중치가 한눈에 들어옵니다. 이 파일 자체가 신호용 님의 **'커리어 전략 문서'** 역할을 합니다.
- **How (어떻게 동작하는가?):**
  - `core_skills`, `general_backend` 등은 **그룹명**입니다.
  - `weight`: 각 그룹의 **중요도(가중치)**입니다. `core_skills`의 3.0은 `devops_infra`의 1.5보다 2배 더 중요하게 평가됩니다.
  - `keywords`: 실제 텍스트와 비교할 **키워드 목록**입니다.

2. 엔진: `data_processor/personalized_job_filter.py` - 판단을 내리는 실행부

이 파일은 설계도(`YAML`)를 읽어와, 실제 채용 공고 데이터를 분석하고 최종 점수를 계산하는 '두뇌의 연산 장치'입니다.

2.1 `__init__(self, ...)`: 엔진의 시동을 걸고, 설계도를 읽는 과정

```
def __init__(self, config_path: str = None):
    # 1. 설정 파일 경로를 찾는다.
    if config_path is None:
        current_dir = Path(__file__).parent.parent
        config_path = current_dir / 'config' / 'job_filter_config.yaml'
    # 2. YAML 파일을 로드하여 self.my_keywords에 저장한다.
    self.my_keywords = self._load_config(str(config_path))
    # 3. 점수 계산에 사용할 그룹 목록과 최대 점수를 미리 계산해둔다. (효율성)
```

```
    self._score_groups = [...]
    self._max_score = sum(...)
```

- **핵심 역할:** `PersonalizedJobFilter()` 객체가 생성되는 순간, `config/job_filter_config.yaml` 파일을 찾아가 그 내용을 통째로 `self.my_keywords` 라는 변수에 저장합니다. 이제 이 객체는 신호용 님의 모든 키워드와 가중치를 알고 있습니다.

2.2 `calculate_relevance_score(...)`: 핵심 연산 로직

이것이 두뇌의 가장 중요한 부분입니다.

```
def calculate_relevance_score(self, job_title: str, job_description: str) ->
Tuple[bool, float]:
    # 1. 준비 작업: 제목과 본문을 합치고 소문자로 변환
    full_text = f"{job_title} {job_description}".lower().strip()

    # 2. 1차 관문 (제외 키워드): '프론트엔드'가 있지만 '백엔드'는 없는가?
    has_exclude = any(word in full_text for word in self.my_keywords['exclude'])
    is_backend_related = any(word in full_text for word in core_keywords)
    if has_exclude and not is_backend_related:
        return False, 0.0 # 즉시 탈락!

    # 3. 2차 관문 (점수 계산)
    score = 0.0
    for group_name in self._score_groups:
        # ...
        matched_count = sum(1 for keyword in keywords if keyword in full_text)
        if matched_count > 0:
            # 3-1. 점수 증폭기: 많이 맞을수록 보너스 점수!
            multiplier = min(1.0 + (matched_count - 1) * 0.2, 1.5)
            # 3-2. 최종 점수 누적: (그룹 가중치 * 보너스)를 score에 더함
            score += base_weight * multiplier

    # 4. 최종 판단: 정규화 및 합격/불합격 결정
    final_score = score / self._max_score # 점수를 0.0 ~ 1.0 사이로 변환
    is_relevant = final_score >= 0.25 # 25% 이상이면 합격!

    return is_relevant, round(final_score, 3)
```

- **핵심 흐름:** '제외 키워드'라는 첫 번째 관문을 통과한 공고만이 '점수 계산'이라는 두 번째 관문에 진입할 자격을 얻습니다. 여기서 각 그룹별로 매칭되는 키워드 수를 세고, 가중치와 보너스를 곱해 최종 점수를 계산합니다. 이 점수를 기준으로 "이 공고는 분석할 가치가 있는가?"(`is_relevant`)를 최종 결정합니다.

3. 관제탑: `main.py` - 모든 것을 지휘하는 지휘관

`main.py`는 이 모든 과정을 순서대로 지휘하는 관제탑입니다.

```
# 1. 크롤러들에게 "데이터를 가져와!" 라고 명령
crawled_jobs = crawler.crawl(...)
job['description'] = crawler.get_job_description(...)

# 2. 필터(두뇌)에게 "가져온 데이터를 분석해!" 라고 명령
job_filter = PersonalizedJobFilter()
is_relevant, score = job_filter.calculate_relevance_score(title, description)

# 3. 필터의 판단에 따라 다음 행동을 결정
if not is_relevant:
    continue # "쓸모없는 데이터군. 버려!"

# 4. AI 분석가에게 "정밀 분석해!" 라고 명령
analysis_result = analyze_job_posting(description)

# 5. 서기에게 "모든 결과를 Notion에 기록해!" 라고 명령
notion.pages.create(...)
```

- **핵심 역할:** `main.py`는 각 부품(크롤러, 필터, 분석가, DB)이 '무엇'을 하는지는 전혀 신경 쓰지 않습니다. 단지 **'언제', '어떤 순서로'** 각 부품을 호출해야 하는지만을 알고, 전체 파이프라인이 물 흐르듯 진행되도록 지휘하는 역할에만 집중합니다. 이것이 바로 **'관심사의 분리(Separation of Concerns)'**라는 매우 중요한 소프트웨어 설계 원칙입니다.
