Chapter 3-1 (개정판): 확장 가능한 크롤러 아키처 설계

Prologue: 레고 블록처럼 조립하는 크롤러

지금까지 우리의 크롤링 로직은 `main.py` 파일 안에 모두 들어있었습니다. 이는 원티드라는 단 하나의 사이트만 상대할 때는 괜찮았지만, 잡코리아, 사람인 등 새로운 사이트를 추가하려면 `main.py`가 점점 더 복잡해지고 유지보수가 어려워지는 구조입니다.

이번 챕터에서는 **객체 지향 프로그래밍(OOP)**의 핵심 개념을 도입하여, 어떤 채용 사이트든 '레고 블록'처럼 쉽게 갈아 끼울 수 있는 유연하고 확장 가능한 크롤러 구조를 설계할 것입니다. 이 경험은 신호용 님의 코드를 한 단계 더 성숙시켜 줄 것입니다.

1. 학습 목표 (Objectives)

- 추상 클래스(Abstract Class)의 개념과 필요성을 이해하고 설명할 수 있다.
- 공통 기능을 가진 부모 클래스(`BaseCrawler`)를 정의하고, 사이트별 특성을 가진 자식 클래스(`WantedCrawler`)가 이를 상속받도록 구현할 수 있다.
- 기존의 절차 지향적 코드를 객체 지향적으로 리팩토링할 수 있다.

---

2. 핵심 개념: 추상 기본 클래스 (Abstract Base Class)

2.1 What: `ABC`란 무엇인가?

`ABC`는 '추상 기본 클래스(Abstract Base Class)'의 약자로, 그 자체로는 완벽한 객체를 만들 수 없는 **'미완성 설계도' 또는 '뼈대'** 같은 것입니다. 파이썬에서는 `abc` 모듈을 통해 이를 구현합니다.

이 설계도에는 두 가지 종류의 메서드가 들어갈 수 있습니다.

- **일반 메서드:** 뼈대에 이미 붙어있는 살처럼, 자식 클래스가 그대로 물려받아 사용할 수 있는 완성된 기능입니다. 우리 프로젝트에서는 모든 크롤러가 공통으로 사용할 `_setup_driver()` 가 여기에 해당합니다.
- **추상 메서드 (`@abstractmethod`):** "이 위치에는 반드시 이런 기능이 들어가야 합니다!"라고 **이름만 정해둔 빈 공간**입니다. 설계도를 물려받는 자식 클래스가 **반드시 직접 채워 넣어야만 하는 '의무 조항'**입니다. 우리 프로젝트에서는 사이트마다 로직이 완전히 다른 `crawl()` 이 여기에 해당합니다.

`ABC`를 상속받는 클래스는, 이 '의무 조항'인 추상 메서드를 모두 구현하지 않으면 에러가 발생하여 객체를 생성할 수 없습니다.

2.2 Why: '미완성 설계도'를 왜 굳이 사용하는가?

불편하게 의무 조항까지 만들면서 `ABC`를 쓰는 이유는, 여러 개발자가 함께 만들거나 오랫동안 유지보수해야 하는 **'복잡한 시스템'을 '질서 정연하게' 만들기 위해서**입니다.

1. **규칙 강제 및 일관성 확보:** "모든 크롤러는 반드시 `crawl()`이라는 이름의 실행 메서드를 가져야 한다"는 규칙을 '강제'할 수 있습니다. 이 덕분에 A 개발자가 만든 크롤러는 `start_crawling()`으로, B 개발자가 만든 크롤러는 `run_scraper()`로 만드는 식의 혼란을 원천적으로 방지합니다. **모든 크롤러의 사용법이 통일**되어 예측 가능하고 관리하기 쉬운 코드가 됩니다.

2. **실수 방지 (Human Error Prevention):** 개발자가 새로운 크롤러를 만들 때, 가장 핵심적인 기능인 `crawl()` 메서드를 깜빡하고 구현하지 않는 실수를 막아줍니다. "사장님, 자동차 설계도를 드렸는데 바퀴 만드는 걸 깜빡하셨네요?" 같은 상황이 발생하지 않도록 시스템이 강제하는 것입니다.
3. **코드의 재사용성 및 확장성 증대:** WebDriver 설정과 같이 모든 크롤러에 공통으로 필요한 기능은 `BaseCrawler`에 한번만 만들어두면 됩니다. 새로운 '사람인 크롤러'를 만들 때, 우리는 '사람인'에만 해당하는 데이터 추출 로직에만 집중하면 되고, 나머지 공통 기능은 그대로 물려받아 사용하므로 **개발 속도가 빨라지고 코드가 간결**해집니다.

2.3 How: 우리의 '채용 공고 크롤러' 프로젝트에는 어떻게 적용되는가?

`ABC`가 있고 없고의 차이를 `main.py`의 관점에서 비교하면 그 강력함을 바로 체감할 수 있습니다.

`ABC`를 적용했을 때 (After):

`BaseCrawler`라는 '설계도'를 통해 "모든 크롤러는 `crawl()`을 가지고 있다"는 **'계약'이 보장**됩니다. `main.py`는 더 이상 각 크롤러의 개별 사정을 알 필요가 없습니다.

```
# main.py (ABC 적용 후)
from crawlers.wanted_crawler import WantedCrawler
from crawlers.jobkorea_crawler import JobKoreaCrawler # 나중에 추가될 크롤러

# 크롤러들을 리스트에 담기만 하면 됨
crawlers = [
    WantedCrawler(),
    JobKoreaCrawler() # 나중에 사람인, 잡플래닛 크롤러도 이 리스트에 추가
]

all_jobs = []
for crawler in crawlers: # 루프를 돌며 일관된 방식으로 호출
    try:
        # 어떤 크롤러 객체든, 그냥 .crawl()만 호출하면 된다!
        all_jobs.extend(crawler.crawl())
    finally:
        crawler.close_driver()
```

이 구조는 **훨씬 간결하고, 우아하며, 확장성이 뛰어납니다.** 새로운 '사람인 크롤러'를 만들어 저 `crawlers` 리스트에 추가하기만 하면, `main.py`의 다른 코드는 단 한 줄도 수정할 필요가 없습니다. 이것이 바로 우리가 `ABC`를 통해 얻으려는 핵심 가치입니다.

[Mission 9] 크롤러 모듈화 및 추상화

이제 이론을 실제로 적용해볼 시간입니다. 아래 단계를 따라 기존 코드를 새로운 구조로 리팩토링해봅시다.

> **Step 1: 파일 구조 변경**
>
> `web-crowler` 폴더 안에 `crawlers` 라는 새 폴더를 만듭니다. 이 폴더는 앞으로 우리가 만들 모든 크롤러 모듈을 담는 공간이 될 것입니다. 그리고 그 안에 `__init__.py` (빈 파일), `base_crawler.py`, `wanted_crawler.py` 파일을 생성합니다.

```
web-crowler/
├── crawlers/
│   ├── __init__.py
│   ├── base_crawler.py  # 크롤러 설계도 (ABC)
│   └── wanted_crawler.py  # 원티드 전용 크롤러 (자식 클래스)
├── main.py
└── ...
```

Step 2: `base_crawler.py` 작성 (설계도 만들기)

모든 크롤러의 공통 부모가 될 추상 클래스를 정의합니다.

```
# crawlers/base_crawler.py

from abc import ABC, abstractmethod
from selenium import webdriver
from selenium.webdriver.chrome.service import Service as ChromeService
from webdriver_manager.chrome import ChromeDriverManager
from selenium.webdriver.chrome.options import Options

class BaseCrawler(ABC):
    """모든 크롤러가 상속받아야 하는 추상 기본 클래스"""

    def __init__(self, base_url):
        self.base_url = base_url
        self.driver = self._setup_driver()

    def _setup_driver(self):
        """Selenium WebDriver를 설정하고 반환합니다. (공통 기능)"""
        chrome_options = Options()
        chrome_options.add_argument("--headless")
        user_agent = "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/123.0.0.0 Safari/537.36"
        chrome_options.add_argument(f"user-agent={user_agent}")
        chrome_options.add_argument("--no-sandbox")
        chrome_options.add_argument("--disable-dev-shm-usage")
        service =
ChromeService(executable_path=ChromeDriverManager().install())
        driver = webdriver.Chrome(service=service, options=chrome_options)
        return driver

    @abstractmethod
    def crawl(self, keyword: str):
        """
        크롤링 프로세스를 시작하는 메인 메서드. (의무 조항)
        이 메소드는 자식 클래스에서 반드시 구현해야 합니다.
        반환 값은 dict를 담은 list 형태여야 합니다. e.g. [{'title': ...,
'company': ...}, ...]
        """
        pass
```

```
    def close_driver(self):
        """드라이버를 안전하게 종료합니다. (공통 기능)"""
        if self.driver:
            self.driver.quit()
```

Step 3: `wanted_crawler.py` **작성 (원티드 크롤러 구현)**

`BaseCrawler`를 상속받아 원티드에 특화된 로직을 구현합니다.

```
# crawlers/wanted_crawler.py

import time
from selenium.webdriver.common.by import By
from selenium.webdriver.common.keys import Keys
from bs4 import BeautifulSoup
from .base_crawler import BaseCrawler # .base_crawler는 같은 폴더에 있는
base_crawler.py를 의미

class WantedCrawler(BaseCrawler):
    """원티드 사이트 크롤러"""

    def __init__(self):
        # 부모 클래스(__init__)를 호출하여 base_url을 전달
        super().__init__("https://www.wanted.co.kr/")

    def crawl(self, keyword: str = '백엔드'):
        """
        BaseCrawler의 의무 조항을 실제로 구현하는 부분.
        원티드 채용 정보를 크롤링하여 list of dict 형태로 반환합니다.
        """
        print("원티드 크롤링을 시작합니다...")
        self.driver.get(self.base_url)
        time.sleep(2)

        # --- 기존 main.py에 있던 원티드 크롤링 로직 시작 ---
        search_button = self.driver.find_element(By.CSS_SELECTOR,
"button[data-attribute-id='gnb']")
        search_button.click()
        time.sleep(1)

        search_input = self.driver.find_element(By.CSS_SELECTOR,
'input.SearchInput_SearchInput__R6jwT')
        search_input.send_keys(keyword)
        search_input.send_keys(Keys.ENTER)
        time.sleep(2)

        last_height = self.driver.execute_script('return
document.body.scrollHeight')
        while True:
            self.driver.execute_script('window.scrollTo(0,
document.body.scrollHeight);')
            time.sleep(2)
```

```
            new_height = self.driver.execute_script('return
document.body.scrollHeight')
            if new_height == last_height:
                break
            last_height = new_height
        # --- 기존 로직 끝 ---

        # 데이터 추출 및 가공
        job_data = []
        soup = BeautifulSoup(self.driver.page_source, 'lxml')
        job_cards = soup.select("div[role='listitem'] a")

        for card in job_cards:
            title = card.select_one("strong[class*='JobCard_title']").text
            company_name =
card.select_one("span[class*='CompanyName']").text
            link = "https://www.wanted.co.kr" + card['href']
            # 출처(source) 정보를 추가하여 반환
            job_data.append({"title": title, "company": company_name,
"link": link, "source": "Wanted"})

        print(f"원티드에서 총 {len(job_data)}개의 공고를 찾았습니다.")
        return job_data
```

Step 4: `main.py` 리팩토링 (간결해진 조율자)

이제 `main.py`는 각 크롤러를 '호출'하고, 그 결과를 받아 Notion에 저장하는 '조율자(Orchestrator)' 역할만 수행하게 되어 훨씬 깔끔해집니다.

```
# main.py (수정 후 예시)

import os
from dotenv import load_dotenv
import notion_client
from crawlers.wanted_crawler import WantedCrawler # WantedCrawler를 임포트

load_dotenv()

# 1. Notion API 설정 (기존과 동일)
NOTION_API_KEY = os.environ.get("NOTION_API_KEY")
NOTION_DATABASE_ID = os.environ.get("NOTION_DATABASE_ID")
notion = notion_client.Client(auth=NOTION_API_KEY)

# 2. 크롤러 실행
all_jobs = []
wanted_crawler = WantedCrawler()
try:
    wanted_jobs = wanted_crawler.crawl(keyword='백엔드')
    all_jobs.extend(wanted_jobs)
finally:
    # 크롤링이 성공하든 실패하든 드라이버는 항상 종료되도록 finally 구문 사용
```

```
    wanted_crawler.close_driver()

# 여기에 나중에 JobKoreaCrawler, SaraminCrawler 등을 추가할 수 있습니다.
# jobkorea_crawler = JobKoreaCrawler()
# all_jobs.extend(jobkorea_crawler.crawl())

# 3. Notion에 저장 (데이터 형식 변경에 주의)
print(f"총 {len(all_jobs)}개의 채용 공고를 찾았습니다. Notion에 저장을 시작합니다...")
for job in all_jobs:
    title = job['title']
    company = job['company']
    link = job['link']
    # source = job['source'] # (나중에 Notion DB에 '출처' 컬럼 추가 후 사용)

    # Notion DB에 페이지 생성 로직 (기존과 유사)
    # ...

print("모든 작업이 완료되었습니다.")
```