4장. 스트림 소개

우리가 기대한 컬렉션 연산은?

첫번째, 무엇을 원하는지는 명시하되, 세부 방법은 명시하지 않는다.

무엇을 원하는지는 명시를 하되 세부적인 명세는 피하는 것이다!

저자가 예로 들기를, `SELECT nAME FROM dishes WHERE calorie <= 400` 이라는 문장으로 400 이하의 메뉴를 조회하길 원한 다는 것을 클라이언트가 데이터베이스에게 요청하지만 세부적인 방법에 대해서는 명시하지 않는다. 무엇을 원하는지만 명시할 뿐 명확한 방법에 대해서는 적지 않기를 바란다는 것이다.

두번째, 멀티 코어 아키텍처로 더 빨리 결과를 산출한다.

이전보다 더욱 많아진 코어를 활용하여 병렬로 빠르게 결과를 산출하기를 기대한다. 그 과정에서 가능하면 간결한 코드를 원한다.

(직접 Thread를 생성하고 synchronized, join 등의 메소드를 통해 구현한 병렬 처리 로직은 구현과 디버깅부터 유지보수까지 모든 게 힘들다!)

위 모든 기대사항의 답은 스트림이다!

스트림이란?

스트림은 자바 8 API 추가된 기능으로, 선언형으로 컬렉션 데이터를 처리할 수 있다. 또한 멀티 쓰레딩 위한 **복잡한** 코드를 작성하지 않아도 간단하게 병렬로 처리하는 어플리케이션을 구현할 수 있다.

전과 후의 차이는?

코드가 간결해지며 가비지 변수(중간 연산 저장 공간)를 사용하지 않아도 되며, 간단히 병렬 처리 되는 코드를 작성할 수 있다.

예를 들어, 전체 메뉴에서 칼로리가 낮은 메뉴를 오름차순으로 산출해낸다고 했을 때 자바 7까지는 FOR-LOOP와 중간 결과를 담기 위한 리스트 저장소가 필요했다.

하지만, 스트림을 사용하면 선언형 코드로 전체 데이터에서 일부 데이터를 필터하고 정렬하는 것을 병렬로 실행할 수 있다.

```
List<Menu> menus = Arrays.asList(new Menu("닭가슴살", 120)
                , new Menu("생연어", 300)
                , new Menu("짜장면", 500));

menus.parallelStream()
        .filter(menu -> menu.getCalorie() >=300)
        .sorted(comparing(Menu::getCalorie)) // default: Descending
        .map(Menu::getName)
        .collect(toList());

// list => [짜장면, 생연어]
```

위 코드에 있는 filter,, sorted, map. collect와 같은 연산은 고수준 빌딩 블록으로 이루어져 있기 때문에 특정 스레딩 모델이 제한되지 않고 어떤 상황에서든 사용할 수 있다. 결과적으로 우리는 데이터 처리 과정을 병렬화하면서 스레드와 락을 걱정할 필요가 없다.

정리하자면 스트림 API의 특징을 아래와 같이 요약할 수 있다.

- **선언형: 더 간결하고 가독성 있음**
- **조리할 수 있음: 유연성이 좋음**
- **병렬화: 처리 성능 향상**

스트림 시작하기

스트림이란?

데이터 처리 연산을 지원하도록 소스에서 추출된 연속된 요소

스트림의 특징은?

파이프라이닝

연속된 데이터에 대한 연산을 연결함으로써 결과를 산출해낼 수 있도록 계속해서 스트림 자신을 반환하도록 하는 특성이다. 그 덕분에 게으름과 쇼트서킷이란 최적화도 얻을 수 있따.

내부 반복

반복자를 이용해 명시적으로 반복하는 컬렉션과 달리 내부 반복을 지원한다.

스트림 vs 컬렉션

기본적인 비교

컬렉션은 현재 자료구조가 포함하는 **모든** 값을 메모리 저장하는 자료구조다.

컬렉션은 생성 때 모든 데이터가 채워진다.(생산자 중심)

컬렉션은 여러번 조회할 수 있다.

스트림은 이론적으로 요청할 때만 요소를 계산하는 고정된 자료 구조다. 실제로, 무한으로 반복되는 스트림을 정의할 수 있다.

스트림은사용자가 데이터를 요청할 때 값을 계산한다.(요청 중심 제조)

스트림은 **한번만 조회**할 수 있다.

외부 반복과 내부 반복

컬렉션은 for-each를 사용해서 사용자가 직접 요소를 반복하는 외부 반복을 한다.

반면에, 스트림은 반복을 알아서 처리하고 결과 스트림값을 어딘가에 저장해주는 내부 반복을 사용한다.

스트림은 함수에 어떤 작업을 수행할지만 지정하면 모든 것이 알아서 처리된다. 반복 과정에 대해 신경쓰지 않고 작업 내용에 집중하면 된다.뿐만 아니라 병렬성을 위한 별도의 코드를 작성하지 않아도 된다.

스트림 연산

스트림은 연결할 수 있는 스트림 연산인 중간 연산과 스트림을 다는 연산인 **최종 연산**으로 구분된다,

중간연산

filter와 sorted 같은 중간 연산은 다른 스트림을 반환한다. 따라서 여러 중간 연산을 연결해 질의를 만들 수 있다. 중간 연산의 중요한 특징은 단말 연산을 스트림 파이프라인이 실행하기 전까지는 아무 연산도 수행하지 않는다는 것, 즉 게으르다**(lazy)**는 것이다.

이러한 게이른 특성 덕에 성능상에 여러 최적화 효과를 얻을 수 있다.

아래 코드를 보자.

```
// Dish.class
public class Dish {
    private final String name;
    private final boolean vegetarian;
    private final int calories;
    private final Type type;

    public Dish(String name, boolean vegetarian, int calories, Type type) {
        this.name = name;
        this.vegetarian = vegetarian;
        this.calories = calories;
        this.type = type;
    }

    public String getName() {
        return name;
    }

    public boolean isVegetarian() {
```

```
        return vegetarian;
    }

    public int getCalories() {
        return calories;
    }

    public Type getType() {
        return type;
    }

    public enum Type { MEAT, FISH, OTHER }

    public static List<Dish> defaultList() {
        return Arrays.asList(
                new Dish("pork", false, 800, Type.MEAT),
                new Dish("beef", false, 700, Type.MEAT),
                new Dish("chicken", false, 400, Type.MEAT),
                new Dish("fries", true, 500, Type.OTHER),
                new Dish("rice", true, 350, Type.OTHER),
                new Dish("fruit", true, 120, Type.OTHER),
                new Dish("pizza", false, 600, Type.OTHER),
                new Dish("prawns", false, 300, Type.FISH),
                new Dish("salmon", false, 450, Type.FISH)
        );
    }
}
```

```
// ShortCircuitTest.class
List<Dish> menus = Dish.defaultList();
List<String> names = dishes.stream()
        .filter(dish -> {
            System.out.println("filtering   " + dish.getName());
            return dish.getCalories() > 300;
        })
        .map(dish -> {
            System.out.println("mapping " + dish.getName());
            return dish.getName();
        })
        .limit(3L)
        .collect(toList());

System.out.println(names);
```

위 코드를 실행해보면 filtering과 mapping이 콘솔에 단 3회 밖에 출력되지 않는다. 이는 limit 연산과 쇼트서킷 덕분이다. 또한 filtering과 map은 다른 연산이지만 한 과정으로 병합되는데 이를 루프 퓨전이라고 한다.

최종연산

스트림 파이프라인에서 결과를 도출한다.보통 최종 연산에 의해 List, Integer, void 등 스트림ㅇ 이외의 결과가 반환된다.