

4. 스트림 소개 - 스템 다리미는 아는데 스트림은 뭘까



Chapter 4. 스트림 소개 - 스템 다리미는 아는데 스트림...?



4.1 스트림이란?

Stream은 자바 8 API에 새로 추가된 핵심 기능이다. 스트림을 이용하면 **선언형**으로 컬렉션 데이터를 처리할 수 있다.

💡 핵심 아이디어

- 데이터 컬렉션 반복을 멋지게 처리하는 기능
- 멀티 스레드 코드를 구현하지 않아도 데이터를 투명하게 병렬로 처리



기존 코드 vs 스트림 코드 비교

◆ 기존 자바 7 방식

```
// 1. 400칼로리 이하 요리 필터링
List<Dish> lowCaloricDishes = new ArrayList<>();
for(Dish dish : menu){
    if(dish.getCalories() < 400) {
        lowCaloricDishes.add(dish);
    }
}

// 2. 칼로리 순 정렬
Collections.sort(lowCaloricDishes, new Comparator<Dish>(){
    public int compare(Dish dish1, Dish dish2){
        return Integer.compare(dish1.getCalories(), dish2.getCalories());
    }
});

// 3. 요리 이름만 추출
List<String> lowCaloricDishName = new ArrayList<>();
for(Dish dish : lowCaloricDishes){
    lowCaloricDishName.add(dish.getName());
}
```

✨ 최신 스트림 방식

```
List<String> lowCaloricDishName = menu.stream()
    .filter(d -> d.getCalories() < 400)    // 400칼로리 이하의 요리 선택
    .sorted(comparing(Dish::getCalories))  // 칼로리로 요리 정렬
```

```
.map(Dish::getName)
.collect(toList());
```

```
// 요리명 추출
// 모든 요리명을 리스트로 저장
```

스트림 파이프라인

스트림 연산을 연결해서 **파이프라인**을 형성한다:

```
menu → filter → sorted → map → collect
```

⚠ **주의사항:** 스트림은 매우 비싼 연산이다.

스트림 API의 특징

- **선언형:** 더 간결하고 가독성이 좋다
- **조립 가능:** 유연성이 뛰어나다
- **병렬화:** 성능 향상 가능

4.2 스트림 시작하기

자바 8 컬렉션에는 스트림을 반환하는 `stream()` 메소드가 추가되었다.

스트림의 정의

스트림이란 데이터 처리 연산을 지원하도록 소스에서 추출된 **연속된 요소**이다.

스트림의 핵심 개념

1 연속된 요소

- 컬렉션과 마찬가지로 연속된 집합의 인터페이스를 제공한다
- **컬렉션:** 시간과 공간의 복잡성과 관련된 요소 저장 및 접근 연산이 주를 이룬다
- **스트림:** filter, sorted 등 표현 계산식이 주를 이룬다
- 간단히 말해 → 컬렉션은 **데이터**, 스트림은 **계산**

2 소스

- 리스트로 스트림을 만들면 스트림의 요소는 리스트 요소와 같은 순서를 유지한다

3 데이터 처리 연산

- 일반적으로 지원하는 연산과 DB와 비슷한 연산을 지원한다
- filter, map, reduce 등으로 데이터를 조작한다
- 순차 혹은 병렬로 실행이 가능하다

★ 스트림의 특징

🔗 파이프라이닝

- 스트림 연산끼리 연결해서 거대한 파이프라인을 구성할 수 있도록 스트림 자신을 반환한다
- 게으름(lazy), 쇼트서킷 같은 최적화도 얻을 수 있다

🔄 내부 반복

- 스트림은 내부 반복 방식을 사용한다

📋 스트림 사용 예제

```
List<String> threeHighCaloricDishNames = menu.stream() // menu가 데이터 소스
    .filter(dish -> dish.getCalories() > 300) // 데이터 처리 연산 시작
    .map(Dish::getName)
    .limit(3)
    .collect(toList()); // 데이터 처리 연산 끝
```

- collect() 제외한 모든 연산은 서로 파이프라인을 형성할 수 있도록 stream을 반환한다
- collect() 연산으로 파이프라인을 처리해서 결과를 반환한다

😬 개인적인 궁금증

💡 findAll() vs SQL 조건 쿼리 비교

질문: findAll()로 모두 가져온 뒤 Stream filter로 거르는 것과 SQL 문 혹은 JPA 쿼리에서 조건을 걸어 가져오는 것의 차이점은?

🏆 DB에서 먼저 거르는 것이 좋은 이유

1) 짐은 최대한 적게 📦

- DB에 조건에 맞는 것만 달라고 하면 필요한 몇 줄만 네트워크를 타고 온다
- 반대로 findAll()로 다 끌어오면 안 쓸 데이터까지 다 날아와서 트래픽 및 메모리가 낭비된다

2) 검색 전문가 DB 🎯

- 인덱스, 최적화 같은 특수 장비를 풀가동해서 찾아준다
- 애플리케이션은 자바코드로 비교한다

🎭 비유로 이해하기

DB 필터: 서점 직원에게 "A책 3권만 주세요" 요청

findAll() 후 stream: 서점의 모든 책을 집으로 가져오고 A책을 골라내기

💡 그럼 Stream은 언제 사용할까?

- 이미 메모리에 있는 작은 리스트를 다룰 때 → 캐시에 올라온 200개 객체 가공

- DB에서 1차로 줄여오고 추가 가공이 필요할 때

4.3 스트림과 컬렉션

컬렉션과 스트림 모두 연속된(순차적으로 값에 접근) 요소의 형식의 값을 저장하는 자료구조 인터페이스이다.

핵심 차이점: 데이터를 언제 계산하느냐

컬렉션

- 현재 자료구조에 포함하는 **모든 값을 메모리에 저장**하는 자료구조이다
- 컬렉션의 모든 요소는 컬렉션에 추가하기 전에 계산되어야 한다
- 연산을 수행할 때마다 컬렉션의 모든 요소를 메모리에 저장해야 한다

스트림



- 이론적으로 **요청할 때만 요소를 계산**하는 자료구조이다
- 요소를 추가하거나 제거할 수 없다
- 사용자가 요청하는 값만 스트림에서 추출할 수 있다

특징 비교표

구분	Collection	Stream
목적	데이터 저장 및 보관	데이터 처리 및 변환
평가 방식	즉시 평가	지연 평가 (최종 연산 호출 시 실행)
재사용성	재사용 가능	한 번만 소비 가능
반복 방식	외부 반복 사용	내부 반복 사용
병렬 처리	병렬처리 직접 구현	<code>parallelStream()</code> 으로 쉬운 병렬처리
수정 가능성	요소 추가 및 삭제 가능	요소 추가 및 삭제 불가능 (불변성)

4.3.1 한 번만 탐색 가능

반복자와 마찬가지로 스트림도 **한 번만 탐색**이 가능하다. 한 번 탐색한 요소를 다시 탐색하려면 데이터 소스에서 새로운 스트림을 만들어야 한다.

```
List<String> title = Arrays.asList("Java", "Stream", "API");
Stream<String> s = title.stream();
s.forEach(System.out::println); //  출력 가능
s.forEach(System.out::println); //  IllegalStateException 발생
```

4.3.2 외부 반복 vs 내부 반복

외부 반복 (컬렉션)

컬렉션 인터페이스를 사용하려면 사용자가 직접 요소를 반복해야 한다.

```
// 외부 반복 - 명령형
List<String> names = new ArrayList<>();
Iterator<Dish> iterator = menu.iterator();
while(iterator.hasNext()){
    Dish dish = iterator.next();
    names.add(dish.getName());
}
```

특징: 하나씩 꺼내어 처리, 지연처리 X, 병렬처리(수동), 순차처리 적합, 디버깅 쉬움, 재사용 가능

내부 반복 (스트림)

스트림 라이브러리가 알아서 처리하고 결과 스트림 값을 어딘가에 저장해준다.

```
// 내부 반복 - 선언형
List<String> names = menu.stream()
    .map(Dish::getName)           // getName메소드로 파라미터화
    .collect(toList());          // 파이프라인 실행
```

특징: 파이프라인 구성 + 최종 연산 시 한번에 처리, 지연처리 O, 병렬처리 O, 재사용 X, 디버깅 어렵다

4.4 스트림 연산

```
List<String> names = menu.stream()
    .filter(dish -> dish.getCalories() > 300) // 중간 연산
    .map(Dish::getName)                       // 중간 연산
    .limit(3)                                 // 중간 연산
    .collect(toList());                      // 최종 연산
```

연산의 분류

```
menu → filter → map → limit → collect
      [   중간 연산들   ]   [최종 연산]
```

- **filter, map, limit:** 서로 연결되며 파이프라인을 형성한다
- **collect:** 파이프라인을 실행하고 닫는다

중간 연산

- 하나씩 체이닝된다
- 각각 새로운 stream을 반환하여 하나의 파이프라인의 일부가 된다

🎯 최종 연산

- 호출해야 전체 파이프라인이 평가된다

⚡ 4.4.1 중간 연산

중간 연산은 **다른 스트림을 반환**한다. 여러 중간 연산을 연결해서 질의를 만들 수 있다.

💡 **핵심**: 단말 연산을 스트림 파이프라인에 실행하기 전까지는 아무 연산도 수행하지 않는다. 즉, **게으르다**는 것이다.

🔧 Stream 파이프라인 구성 요소

- 중간 연산들이 순서대로 체이닝된다
- 최종 연산을 호출하면 체인 전체가 실행된다

⚡ 병렬 처리 동작 방식

- 내부적으로 ForkJoinPool 기반의 스레드 풀이 동작한다
- 병렬적으로 처리될 수 있게 작업을 분할한다 → 분할 정복 알고리즘 기반

🔍 **지연 평가 (Lazy Evaluation)**: 중간 연산은 실제로 실행되지 않고 내부적으로 연산정보만 저장하는 반면, 최종 연산은 등록된 연산을 한번에 수행한다.

🎯 4.4.2 최종 연산

스트림 파이프라인에서 **결과를 도출**한다. 보통 최종 연산에 의해 `List`, `Integer`, `void` 등 **스트림 이외의 결과를 반환**한다.

📋 4.4.3 스트림 이용 과정 요약

🎯 3단계 과정

1. 질의를 수행할 데이터 소스
2. 스트림 파이프라인을 구성할 중간 연산 연결
3. 스트림 파이프라인을 실행하고 결과를 만들 최종 연산

🔧 주요 연산들

중간 연산: `filter`, `map`, `limit`, `sorted`, `distinct` 등

최종 연산: `forEach`, `count`, `collect` 등

🎉 마무리

스트림 API는 자바 8의 핵심 기능 중 하나로, 데이터 처리를 더욱 간결하고 효율적으로 만들어준다. 선언형 프로그래밍의 장점을 활용해 더 읽기 쉬운 코드를 작성할 수 있다.

💡 **기억하자:** 스트림은 한 번만 소비 가능하고, 지연 평가를 통해 최적화된 성능을 제공한다!