모던 자바 인 액션 1장, 2장 정리

황의찬

목차

Stream Api Convention

http://blog.marcinchwedczuk.pl/java-streams-best-practices

```
// BAD CODE:
strings.stream().filter(s -> s.length() > 2).sorted()
        .map(s -> s.substring(0, 2)).collect(Collectors.toList());


// GOOD CODE:
strings.stream()
        .filter(s -> s.length() > 2)
        .sorted()
        .map(s -> s.substring(0, 2))
        .collect(Collectors.toList());
```

역사의 흐름은 무엇인가?

Arrays.asList() vs List.of()

Java에서 Array를 List로 변환하기 위해서는 Arrays.asList(array)를 사용했습니다.
Java 9 버전부터 List.of(array) 라는 새로운 팩토리 메서드를 도입했습니다.

차이점은 무엇일까요?

변경 가능 여부

```
public static void main(String[] args) {
  List<Integer> arrayList = Arrays.asList(1, 2, null);
  arrayList.set(1, 10); // OK

  List<Integer> list = List.of(1, 2, 3);
  list.set(1, 10); // Fails with UnsupportedOperationException

}
```

Arrays.asList()는 ArrayList를 반환하고, set 등이 구현되어있어 변경이 가능합니다(Mutable).
반면에 List.of()는 ListN이라는 타입의 객체를 반환하는데, 이는 불변 객체입니다(Immutable object). 따라서 수정할 수 없습니다.

참고로 Arrays.asList() 가 반환하는 ArrayList 타입은 java.util.ArrayList가 아니라 Arrays 내부 클래스입니다. add와 remove는 구현되어 있지 않습니다.

역사의 흐름은 무엇인가?

Arrays.asList() vs List.of()

Null 허용 여부

```
List<Integer> arrayList2 = Arrays.asList(1, 2, null); // OK
List<Integer> list2 = List.of(1, 2, null); // Fails with NullPointerException

arrayList2.contains(null); // return false
list2.contains(null); // Fails with NullPointerException
```

Arrays.asList()는 null을 허용합니다. List.of()는 반환 객체가 생성될 때, 내부적으로 파라미터들에 대한 null 체크를 하고 null을 허용하지 않습니다.

List.of()로 반환된 객체의 contains 의 경우 null이 들어오면 NPE(Null pointer Exception) 이 발생합니다.

역사의 흐름은 무엇인가?

Arrays.asList() vs List.of()

참조/비참조

```java
Integer[] array = {1, 2};
List<Integer> arrayList = Arrays.asList(array);
array[0] = 100;
System.out.println(arrayList); //[100, 2]

Integer[] array2 = {1, 2};
List<Integer> list = List.of(array2);
array2[0] = 100;
System.out.println(list); //[1, 2]
```

Arrays.asList()의 반환 객체로 add와 remove를 구현할 수 없는 이유는 **참조**로 동작하기 때문입니다.
Arrays.asList()는 참조를 사용하기 때문에 배열의 값이 변경되면 list에도 영향이 갑니다.

List.of(array)의 결과는 값을 기반으로 독립적인 객체를 만들기 때문에 참조가 일어나지 않습니다.

역사의 흐름은 무엇인가?

Arrays.asList() vs List.of()

메모리 사용

Arrays.asList()는 List.of()보다 힙에 더 많은 개체를 생성하기 때문에 더 많은 오버헤드 공간을 차지합니다. 따라서 단지 값 요소가 필요한 경우라면 List.of()가 적합합니다.

변환

예를 들어 Array를 ArrayList 또는 HashSet 등으로 변환하기 위해서는, 참조나 변경 가능 여부는 상관없고 요소만 알면 됩니다. 이때는 List.of()가 적합합니다.

```
List<Integer> integerList = new ArrayList<>(List.of(array2));
Set<Integer> integerSet = new HashSet<>(List.of(array2));
```

역사의 흐름은 무엇인가?

Arrays.asList() vs List.of()

요약 : 그래서 어떤 걸 써야할까..?

> Arrays.asList()는 크고 동적인 데이터에 사용하고, List.of()는 작고 변경되지 않는 데이터의 경우 사용합니다.
>
> List.of()는 필드 기반 구현이 있고, 내부적으로 힙 공간을 덜 사용하기에 요소 자체가 필요할 때 사용합니다.
>
> Arrays.asList()는 변경이 가능하고 thread safety하지 않고, List.of()는 불변이고 thread safety합니다.
>
> Arrays.asList()는 null 요소를 허용하고 List.of()는 null 요소를 허용하지 않습니다.
>
> Arrays.asList(), List.of() 모두 크기는 변경할 수 없습니다. 크기를 바꾸려면 Collections을 생성해서 요소의 값을 옮겨야 합니다.

역사의 흐름은 무엇인가?

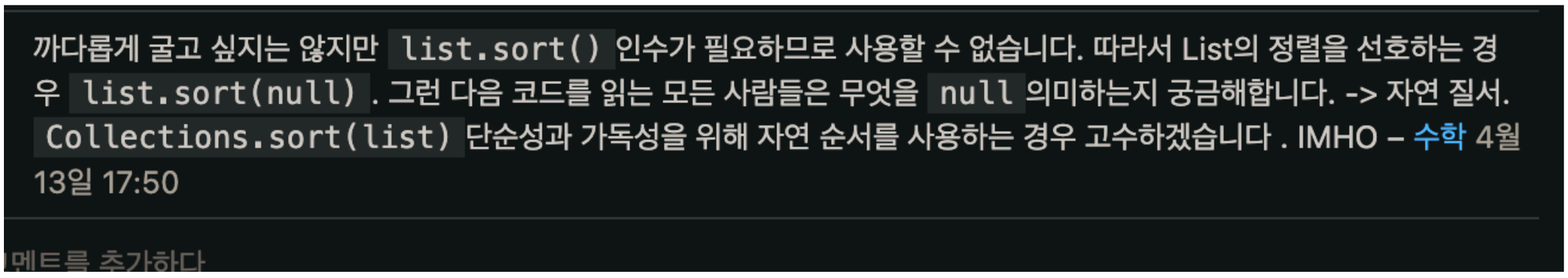
Collections.sort vs List.sort

Collection.sort Java 1.2부터 존재했습니다.

Java8에서 디폴트 메서드의 등장으로 List 인터페이스에 다음과 같은 디폴트 메서드 정의가 추가되었기 때문에 List에 직접 sort 메서드를 호출할 수 있습니다.

```
default void sort(Comparator<? super E> c) {
    Collections.sort(this, c);
}
```

따라서 Java8 이전에는 List를 구현하는 모든 클래스가 sort를 구현해야 했지만 Java8부터는 디폴트 sort를 구현하지 않아도 됩니다.



IntelliJ의 ListSortTest

역사의 흐름은 무엇인가?

Comparable vs Comparator

https://st-lab.tistory.com/243

간단하게 알아보면, Comparable 과 Comparator 모두 인터페이스(interface)입니다.
즉, Comparable 혹은 Comparator 을 사용하고자 한다면 인터페이스 내에 선언된 메서드를 '**반드시 구현**' 해야 합니다.

Comparable - compareTo(T o) 메서드

Comparator - compare(T o1, T o2) 메서드

자바 함수(48p~)

method reference 예제(50p)

```
emp.stream()
    .sorted(comparing(Employee::getFirstName) //비교할 값을 결정하는 Function 을 파라미터로 받아 Comparator 생성
        .thenComparing(Employee::getLastName))
    .forEach(System.out::println);
```

스트림

컬렉션을 필터링할 수 있는 가장 빠른 방법은..?

컬렉션을 **스트림**으로 바꾸고, 병렬로 처리한 다음에, 리스트로 다시 복원하는 것입니다.

```
List<Apple> heavyApples = inventory.stream()
    .filter((Apple a) → a.getWeight() > 150)
    .collect(toList());
```

스트림과 **람다** 표현식을 이용하면 **'병렬성을 공짜'**로 얻을 수 있으며 리스트에서 무거운 사과를 순차적으로 또는 병렬로 필터링 할 수 있습니다.(책 59p)

collect 는 Stream의 데이터 변형 등의 처리를 하고 원하는 자료형으로 변환해줍니다.

그런데 아래의 코드도 가능합니다..

```
List<Apple> heavyApplesWithStreamToList =
    inventory.stream()
    .filter(Apple::isGreenApple).toList();
```

스트림

Stream.toList 로 Stream.collect(toList())를 대체해도 되는 걸까?

```
List<Apple> heavyApples =
    inventory.stream()
    .filter((Apple a) → a.getWeight() > 150)
    .collect(Collectors.toList());
```

사실 Collectors.toList()인데, 대부분 static import 해서 사용해왔습니다.

Stream.toList()는 JDK 16에서 생겼습니다. 기존의 Collectors.toList() 를 toList() 메서드 하나로 줄여 더 간결화시켰습니다.

그렇다면 **차이점**은 무엇일까요?

collect(Collectors.toList())는 **ArrayList**를 반환합니다.

toList()는 **Collectors.UnmodifiableList** 또는 **Collectors.UnmodifiableRandomAccessList**를 반환합니다.

이름에서 나타나는 것처럼 수정이 가능한 구현체와 수정이 불가능한 구현체라는 것입니다.

스트림

Stream.toList 로 Stream.collect(toList())를 대체해도 되는 걸까?

그럼 **JDK 17**을 사용하는 프로젝트에서 Stream.collect(toList()) 를 Stream.toList() 로 대체해도 될까요?

다음 코드를 생각해봅시다.

```
public void someMethod() {
        var peopleName = people.stream()
                               .map(Person::name)
                               .toList(); // UnmodifiableList
        someBusinessLogic(peopleName);

        ...

}

private void someBusinessLogic(List<String> peopleName) {
        peopleName.add("Name that should always be added"); // UnsupportedOperationException
}
```

앞서 말했던 것처럼 Stream.toList()는 UnmodifiableList를 반환하기에 List 자체에 대한 변경이 불가능합니다.
수정에 관련된 메서드들은 UnsupportedOperationException 예외를 던지는데, RuntimeException의 구현체로 런타임 시점에 알 수 있습니다. IDE도 오류를 위 코드에서 오류를 예측하지 못합니다.

스트림

Stream.toList 로 Stream.collect(toList())를 대체해도 되는 걸까?

collect(Collectors.toList())는 이후에 List 수정 로직이 존재한다면 Stream.toList()로 대체하지 못합니다.

```
// Before jdk 17
var peopleName = people.stream()
                       .map(Person::name)
                       .collect(Collectors.toUnmodifiableList());

// jdk 17
var peopleName = people.stream()
                       .map(Person::name)
                       .toList();
```

그럼 위 코드는 가능할까요? 가능하긴 한데 차이가 있긴 있습니다.

collect(Collectors.toUnmodifiableList())의 경우 input list에 **null을 허용하지 않고,**
Stream.toList() 는 null 요소를 **허용합니다**.

스트림

Stream.toList 로 Stream.collect(toList())를 대체해도 되는 걸까?

요약 : 그래서 어떤 걸 써야할까..?

collect(Collectors.toList())
✔수정이 **가능** 합니다.
✔ Null 값이 허용됩니다.

collect(toUnmodifiableList())
✔수정이 **불가능** 합니다.
✔ Null 값이 허용되지 않습니다.

toList()
✔수정이 **불가능** 합니다.
✔ Null 값이 허용됩니다.

이후에 수정해야 할 일이 있을 땐 collect(Collectors.toList())를 사용하는 것이 좋아보입니다.

스트림

깊은 복사(Deep Copy)와 얕은 복사(Shallow Copy) + UnmodifiableList + 방어적 복사

깊은 복사는 **‘실제 값’을 새로운 메모리 공간에 복사하는 것**을 의미합니다.
얕은 복사는 ‘주소 값’을 복사한다는 의미입니다. -> **참조하고 있는 실제값은 같습니다.**

스트림

Stream() vs parallelStream()

parallelStream이 Stream보다 항상 빠른가?

결론부터 말하자면 **당연히 아닙니다.**

스트림 파이프라인을 아무 생각 없이 parallelStream으로 돌리면 안 된다.
성능이 나빠지는 것에 그치지 않고, 응답 불가까지 발생시키는 심각한 장애와 결과에 오류가 있을 수 있다.

ArrayList, HashMap, HashSet, ConcurrentHashMap, 배열, int 범위, long 범위에서 효과가 가장 좋다.
이들 자료구조가 데이터를 원하는 크기로 정확하게 나눌 수 있고, 원소를 순차 실행할 때 참조 지역성이 뛰어나다.(메모리에 연속해서 저장되어 있다.)

파이프라인의 종단 연산이 어떤 것이냐에 따라 성능이 결정된다.
종단 연산 중 가장 parallelStream의 덕을 많이 보는 것은 reduction 작업과 그다음으로는, 조건에 맞으면 바로 반환하는 anyMatch, allMatch, noneMatch 등이 있다.

parallelStream은 성능 최적화 수단으로만 사용한다.
성능 최적화인지 확인하기 위해서는 그 가치를 테스트로 검증한다.
만약 직접 구현한 Stream, Iterable, Collection에 적용한다면, 그에 맞는 spliterator를 재정의하라.

스트림

Stream() vs parallelStream() 결론

결론

parallelStream은 정말 쉽게 stream 병렬 처리를 제공해준다. 세부 설정이나 복잡한 로직 없이 기존에 stream을 쓰듯 사용할 수 있는 편리함까지 제공해주지만, 병렬 처리 결과가 무조건 더 나은 결과를 보장한다고 볼 수는 없다.
처리 성능에 영향을 미치는 부분들, 분할 및 병합 과정에서의 비용, 멀티 스레드 환경에서의 컨텍스트 스위칭 비용 등에 대해 충분히 고려해야 하기 때문에 신중해야 한다. 특정 로직에 대해 성능 개선을 위해 parallelStream을 적용하고자 한다면, 이것이 정말로 성능을 개선시켜줄 수 있는지, 혹 예상치 못한 장애를 발생시키지는 않는지에 대해 충분히 테스트도 진행하고 적용하는 것이 좋을 것 같다.

1장 추가 내용

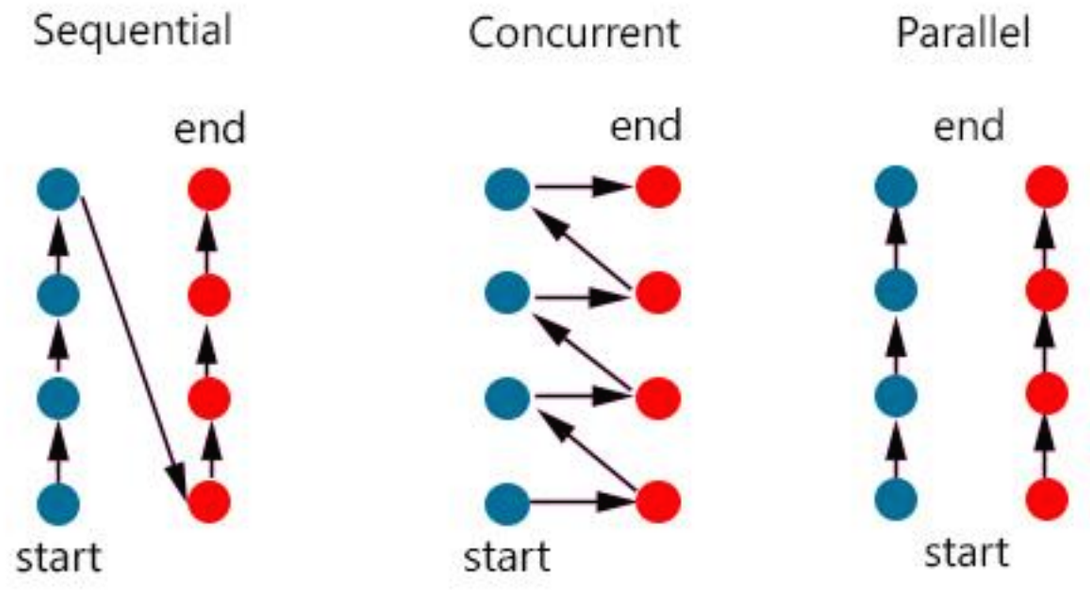
동시성(concurrent)과 병렬성(parallelism)

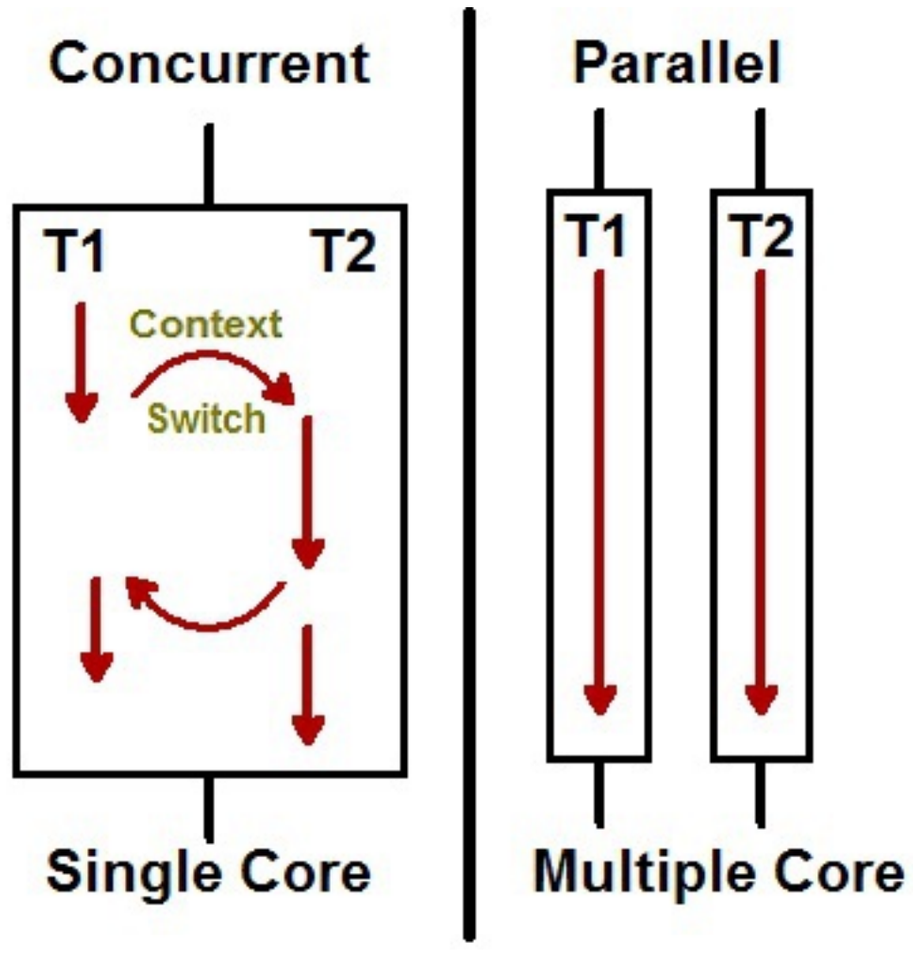
동시성(Concurrency) vs 병렬성(Parallelism)

동시성	병렬성
동시에 실행되는 것 같이 보이는 것	실제로 동시에 여러 작업이 처리되는 것
싱글 코어에서 멀티 쓰레드(Multi thread)를 동작시키는 방식	멀티 코어에서 멀티 쓰레드(Multi thread)를 동작시키는 방식
한번에 많은 것을 처리	한번에 많은 일을 처리
논리적인 개념	물리적인 개념

1장 추가 내용

동시성(concurrent)과 병렬성(parallelism)





Context Switch란?
CPU가 현재 작업 중인 프로세스에서 다른 프로세스로 넘어갈 때 지금까지의 프로세스 상태를 저장하고, 새 프로세스의 저장된 상태를 다시 적재하는 작업을 Context Switch라고 합니다.

감사합니다