1장-3장

1장. 자바 8,9, 10, 11 : 무슨 일이 일어나고 있는가?

개요

멀티코어 CPU 대중화와 같은 하드웨어적인 변화가 자바8에 영향을 미쳤다. CPU 속도 증가에 한계점에 이르며 코어 수를 높이는 쪽으로 하드웨어가 발전해갔는데 이러한 변화에 발맞춰 자바 또한 변화해간 것이다.

또한 빅데이터(테라바이트 이상의 데이터셋)라는 도전에 직면하면서 멀티코어 서버나 클러스터를 이용하여 이를 효율적으로 처리 해야 됐다. 그러나 자바 8 이전의 자바로는 충분히 대응하기가 어려웠다.

물론 자바가 병렬 처리를 위한 API를 완전히 제공하지 않아왔던 것은 아니다.

자바 5에서는 쓰레드 풀, 병렬 실행 컬렉션을 도입했고 자바7에는 포크/조인 프레임워크을 제공했으나 개발자가 사용하기가 쉽지 않았다. 이에 반해 자바8은 **병렬 실행을 새롭고 단순한 방식으로 접근할 수 있는 방법을 제공**한다.

자바 8 설계의 밑바탕에는 세 가지 프로그래밍 개념이 있다.

자바 8 설계의 밑바탕

첫번째, 스트림 처리

스트림은 **한 번에 한 개씩 만들어지는 연속적인 데이터 항목**들의 모임이다.

이론적으로 프로그램은 입력 스트림에서 데이터를 하나씩 읽어들이며 마찬가지로 출력 스트림으로 데이터를 한 개씩 기록한다.

즉, 어떤 프로그램의 출력 스트림은 다른 프로그램의 입력 스트림이다.

그렇기 때문에 스트림을 파이프라인처럼 연결해서 일련의 데이터 가공 과정을 통해 정제하는 것이 가능하고, 스레드라는 복잡한 작업을 사용하지 않아도 **parallel**이란 키워드가 들어간 함수를 통해 병렬성을 얻을 수 있다.(후술하겠지만 사용상의 주의가 필요하다)

두번째, 동작 파라미터화(behavior parameterization)

자바8에서 추가된 동작 파라미터화를 통해 코드 일부를 API를 전달할 수 있게 되었다!

동적 파라미터화는 어떤 로직을 처리할 지 확정하지 않고, 이에 대한 결정을 클라이언트에게 위임한다. 덕분에 중복을 방지할 수도 있고, 소스코드의 유연성도 높일 수 가 있다.

뿐만 아니라 메소드에 처리 내용을 파라미터로 전달해서 메소드 내에 특정 시점에 실행해서 처리하는 것이 가능하다.

자바 8 이전에는 원시 변수나 참조 변수만이 메소드에 전달 가능한 일급 값이었고 메서드 및 처리 로직들은 넘길 수 없는 이급 값이였다.

그러나 동적 파라미터화를 통해 메서드 및 처리 로직들 또한 일급 값이 되었다.

세번째, 병렬성과 공유 가변 데이터

스트림에서 지원하는 parallel로 시작하는 함수를 통해 병렬 처리를 간편하게 할 수 있다. 다만, 안전하고 의도한 대로 동작하려면 사용상의 주의가 필요하다. 이를 위해 책에서 강조하는 것 중 하나가 **공유된 가변 데이터에 대한 접근 지양** 이다.

변화가 가능하다는 것은 유연함을 의미하기도 하지만, 사이드 이펙트를 의미하기도 한다. 그러나 병렬성을 얻으려면 이러한 공유된 가변 데이터에 대해 지양하는 것이 필요하다. 상태 없고 부작용 없는 함수가 되어야 한다는 것이다.

주의를 거쳐 사용되는 병렬 처리는 아주 강력하다.

아래 코드를 실행해보면 콘솔에 출력된 각 테스트의 소요 시간 차이를 비교해볼 수 있다.

```
class StreamExample {
    private static final int TEST_MAX_RANGE = 30_000_000;

    public void example() {
        test("int-stream", () -> {
            IntStream.rangeClosed(1, TEST_MAX_RANGE).sum();
        });

        test("int-parallel-stream", () -> {
            IntStream.rangeClosed(1, TEST_MAX_RANGE).parallel().sum();
        });
    }

    private void test(String label, Consumer<List<Integer>> function) {
        long current = currentTimeMillis();
        function.accept(list);
        long end = currentTimeMillis();
        System.out.println(label + "    " + (end - current));
    }
}
```

디폴트 메소드

기존 인터페이스에 메소드가 추가되면 해당 인터페이스를 구현하는 클래스는 추가된 메소드에 구현이 강제된다. 그러나 이를 원하지 않을 경우 디폴트 메소트로 구현함으로써 회피할 수 있다.

아래는 자바 8의 List 인터페이스에 구현된 sort 메소드다.

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

디폴트 메소드 덕분에 List의 구현체인 ArrayList나 LinkedList는 별도로 sort 메소드를 구현하지 않아도 사용자들은 List 인터페이스에 구현된 sort 메소드를 사용함으로서 정렬을 할 수 있다.

2장. 동적 파라미터화

간단한 예시를 통해 동적 파라미터화가 왜 필요하고 어떻게 코드 개선될 수 있는지 보자.

Niniz 객체를 담고 있는 리스트에서 원하는 니니즈를 찾는 프로그램을 개발한다고 가정하겠다.

Niniz 클래스는 아래와 같다.

```
/* Niniz.java */
public class Niniz {
    private String name;

    private String color;

    private String classification;

    public Niniz(String name, String color, String classification) {
        this.name = name;
        this.color = color;
        this.classification = classification;
    }

    public String getName() {
        return name;
    }

    public String getColor() {
        return color;
    }

    public String getClassification() {
        return classification;
    }
}
```

자바 8 이전에는 Ninizs 리스트에서 초록색인 Niniz를 찾으려면 아래와 같이 했을 것이다.

간단한 로직이여서 한눈에 이해가 되지만, 실제 비즈니스 로직이면 훨씬 복잡했을 것이다.

```
/* Main.java */
public class Main {

    public static void main(String[] args) {
        List<Niniz> ninizs = init();
```

```
        List<Niniz> result = new ArrayList<>();
        for(Niniz niniz : ninizs) {
            if(niniz.getColor().equals("초록색")) {
                result.add(niniz);
            }
        }
    }

    private static List<Niniz> init() {
        return asList(new Niniz("죠르디", "초록색", "파충류")
            , new Niniz("스카피", "분홍색", "포유류")
            , new Niniz("앙몬드", "흰색", "포유류"));
    }
}
```

위 로직을 스트림과 동적 파리미터 화를 사용하면 아래와 같이 개선이 가능하다.

```
public class Main {

    public static void main(String[] args) {
        List<Niniz> ninizs = init();

        List<Niniz> result = ninizs.stream()
                .filter(niniz -> niniz.getColor().equals("초록색"))
                .collect(toList());
    }

    private static List<Niniz> init() {
        return asList(new Niniz("죠르디", "초록색", "파충류")
            , new Niniz("스카피", "분홍색", "포유류")
            , new Niniz("앙몬드", "흰색", "포유류"));
    }
}
```

좀더 짧고 간결해진 것을 알 수 있는데, 대략적인 내용은 초록색인 니니즈를 필터하고 이를 리스트로 변환해 반환하는 코드이다.

앞선 내용에서 스트림은 출력이 곧 입력될 수 있다는 내용이 있는데, 포유류인 니니즈를 필터를 한 후에 흰색인 니니즈를 찾아보도록 하겠다.

```
public class Main {

    public static void main(String[] args) {
        List<Niniz> ninizs = init();

        List<Niniz> result = ninizs.stream()
                .filter(niniz -> niniz.getClassification().equals("포유류"))
                .filter(niniz -> niniz.getColor().equals("흰색"))
```

```
                .collect(toList());
    }

    private static List<Niniz> init() {
        return asList(new Niniz("죠르디", "초록색", "파충류")
            , new Niniz("스카피", "분홍색", "포유류")
            , new Niniz("앙몬드", "흰색", "포유류"));
    }
}
```

filter 메소드에 들어가는 로직을 참조 객체로 선언하고 이를 할당하는 것 또한 가능하다.

```
public class Main {

    private static Predicate<Niniz> mammalFilter = niniz -> niniz.getClassification().equals("포유류");
    private static Predicate<Niniz> whiteFilter = niniz -> niniz.getClassification().equals("포유류");

    public static void main(String[] args) {
        List<Niniz> ninizs = init();

        List<Niniz> result = ninizs.stream()
                .filter(mammalFilter.and(whiteFilter))
                .collect(Collectors.toList());
    }

    private static List<Niniz> init() {
        return asList(new Niniz("죠르디", "초록색", "파충류")
                , new Niniz("스카피", "분홍색", "포유류")
                , new Niniz("앙몬드", "흰색", "포유류"));
    }
}
```

3장. 람다 표현식

```
public class StreamPerformanceTest {

    private static final intTEST_MAX_RANGE= 30_000_000;
    private static List<Integer>list= new ArrayList<Integer>();

    public static void main(String[] args) {
init("init", () -> IntStream.rangeClosed(1,TEST_MAX_RANGE).boxed().collect(toList()));

test("for-loop", (List<Integer> list) -> {
            int result = 0;
            for (int each : list) {
                result += each;
            }
        });

test("int-stream", () -> {
            IntStream.rangeClosed(1,TEST_MAX_RANGE).sum();
        });
```

```
test("int-parallel-stream", () -> {
            IntStream.rangeClosed(1,TEST_MAX_RANGE).parallel().sum();
        });

test("stream", (List<Integer> list) -> {
            list.stream().mapToInt(i -> i.intValue()).sum();
        });

test("parallel-stream-non-unboxing", (List<Integer> list) -> {
            list.parallelStream().mapToInt(i -> i).sum();
        });

test("parallel-stream", (List<Integer> list) -> {
            list.parallelStream().mapToInt(Integer::intValue).sum();
        });

    }

    private static void init(String label, Supplier<List<Integer>> function) {
        long current =currentTimeMillis();
list= function.get();
        long end =currentTimeMillis();
        System.out.println(label + "    " + (end - current));
    }

    private static void test(String label, Consumer<List<Integer>> function) {
        long current =currentTimeMillis();
        function.accept(list);
        long end =currentTimeMillis();
        System.out.println(label + "    " + (end - current));
    }

    private static void test(String label, Runnable function) {
        long current =currentTimeMillis();
        function.run();
        long end =currentTimeMillis();
        System.out.println(label + "    " + (end - current));
    }
}
```

위 코드는 for-loop와 박싱된 Stream, ParallelStream, 그리고 박싱되지 않은 Stream, parallelStream의 성능 비교를 하는 코드이다.

위 코드를 통해 배울 수 있는 것은 아래와 같다.

메소드 참조를 통한 코드 간결화

일부 코드는 메소드 참조를 사용함으로써 람다를 사용하는 것보다 더 간략하게 표현하는 것이 가능하다.

위 코드에서는 `i -> Integer.intValue(i)` 라는 mapToInt함수 내부의 로직이 메소드 참조로 간결화 되어 `Integer::intValue`로 표현되었다.

기본형 클래스의 성능 영향도

그리고 실행해보면 언박싱된 스트림이 박싱된 스트림이나 for-loop보다 빠른 것을 알 수 있는데, 가볍게 생각할 수 있는 박싱/언박싱 유무가 생각 외로 성능에 큰 영향을 줌을 알 수 있다.

IntStream 외에도 IntPredicate, IntConsumer, LongFunction 등 대부분의 기본형 함수형 인터페이스 제공해줌으로 기본형 사용이 가능하면 기본형을 써야할 것이다.

For-loop와 Stream의 성능 비교

for-loop와 박싱된 Stream의 소요시간을 비교해보면 그렇게 큰 차이가 나지 않는 것을 알 수 있는데, 그렇기 때문에 for-loop와 Stream을 두고 고민할 때 parallel을 사용하지 않는 이상 성능이 그 지표가 될 필요는 없으며 Stream은 오히려 lazy processing이 가능하므로 성능과 가독성 모두를 취할 수 있는 선택지가 될 수 있다.

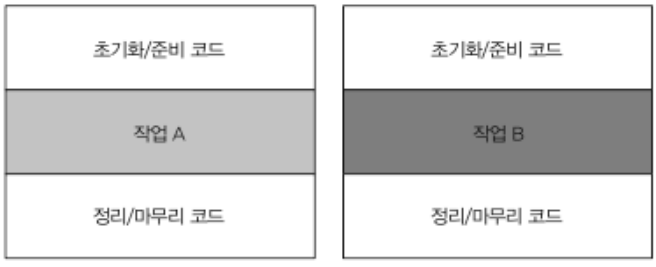
실행 어라운드 패턴

자원처리에 사용하는 순환(반복적인) 패턴은 자원을 열고, 처리한 다음에, 자원을 닫는 순서로 이루어 진다. 서정과 정리 과정을 대부분 비슷하다.

텍스트 파일에서 텍스트를 읽고 자원을 반환하는 코드의 경우에는 아래와 같이 코드가 작성될 수 있는데, 동일 프로젝트 내에서 읽어드리는 파일과 읽어드린 데이터를 처리하는 방법만 변경될 뿐 리더를 열고 닫는 패턴을 비슷할 것이다.

```
public String processFile() throws IOException {
    try (BufferedReader br =
            new BufferedReader(new FileReader("data.txt"))) {
        return br.readLine();  <-- 실제 필요한 작업을 하는 행이다.
    }
}
```

그림 3-2 중복되는 준비 코드와 정리 코드가 작업 A와 작업 B를 감싸고 있다.



위 예시에서는 test란 메소드가 ms를 측정하고 이를 화면에 출력하는 부분은 각 테스트 로직이 모두 동일한 반면 내부에서 실행되는 스트림 로직만 변경되어서 실행 어라운드 패턴을 적용해서 코드를 작성했다.

함수형 인터페이스의 종류

파라미터 타입과 반환 타입에 따라 함수형 인터페이스를 선택할 수 있다. 그 종류는 아래와 같다.

함수형 인터페이스	함수 디스크립터	기본형 특화
Predicate<T>	T -> boolean	IntPredicate, LongPredicate, DoublePredicate
Consumer<T>	T -> void	IntConsumer, LongConsumer, DoubleConsumer
Function<T, R>	T -> R	IntFunction<R>, IntToDoubleFunction, IntToLongFunction, LongFunction<R>, LongToDoubleFunction, LongToIntFunction, DoubleFunction<R>, DoubleToIntFunction, DoubleToLongFunction, ToIntFunction<T>, ToDoubleFunction<T>, ToLongFunction<T>
Supplier<T>	() -> T	BooleanSupplier, IntSupplier, LongSupplier, DoubleSupplier
UnaryOperator<T>	T -> T	IntUnaryOperator, LongUnaryOperator, DoubleUnaryOperator
BinaryOperator<T>	(T, T) -> T	IntBinaryOperator, LongBinaryOperator, DoubleBinaryOperator
BiPredicate<L, R>	(T, U) -> boolean	
BiConsumer<T, U>	(T, U) -> void	ObjIntConsumer<T>, ObjLongConsumer<T>, ObjDoubleConsumer<T>
BiFunction<T, U, R>	(T, U) -> R	ToIntBiFunction<T, U>, ToLongBiFunction<T, U>, ToDoubleBiFunction<T, U>

void 호환 규칙

람다의 바디에 일반 표현식이 있으면 void를 반환하는 함수 디스크립터와 호환된다.(물론 파라미터 리스트도 호환되어야 함) 예를 들어 다음 두 행의 예제에서 List의 add메서드는 Consumter 콘텍스트(T → void)가 기대하는 void 대신 boolean을 반환하지만 유효한 코드다.

```
// list.add는 boolean 반환 값을 가짐
// 그런데, 반환 타입이 void인 Consumer에서 list.add를 써도 컴파일 에러가 발생되지 않음!
```

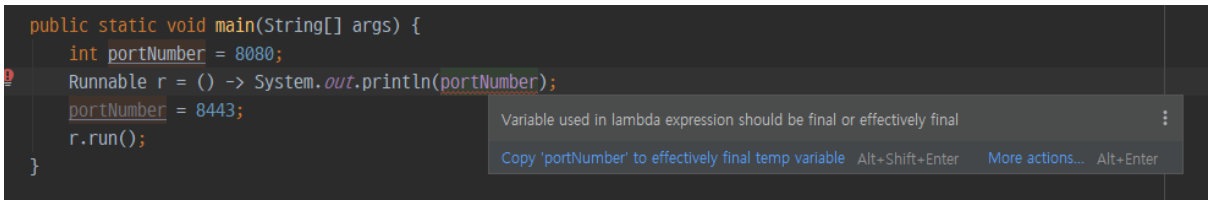
```
Predicate<String> predicate = s -> list.add(s);
Consumer<String> consumer = s -> list.add(s);
```

지역 변수 사용

람다 표현식은 파라미터로 넘겨받은 변수가 아닌 자유 변수라도 사용할 수 있는데 이를 람다 캡쳐링 `capturing lambda` 이라 한다.

그런데, 람다 캡쳐링에도 제약 조건이 있다. 사용될 지역 변수가 final 키워드를 붙인 상수 이거나 상수처럼 사용되어야 한다는 점이다. 만약 이를 무시하고 아래 코드와 같이 값을 변경하려 시도할 경우 컴파일 에러가 발생한다.

아래 코드는 `portNumber = 8443;` 을 제거하면 정상적으로 컴파일이 된다.



지역 변수의 제약

왜 위와 같은 제약이 필요한 걸까? 우선 인스턴스 변수는 힙에 저장되는 반면 지역변수는 스택에 저장된다. 람다에서 지역 변수에 바로 접근할 수 있다는 가정하에 람다가 스레드에 서 실행된다면 변수를 할당한 스레드가 사라져서 변수 할당이 해제 되었는데도 람다를 실행하는 스레드에서는 해당 변수에 접근하려 할 수 있다. 따라서 자바 구현에서는 원래 변수에 접근을 허용하는 것이 아니라 자유 지역 변수의 복사본을 제공한다, 따라서 복사본의 값이 바뀌지 않아야 하므로 지역 변수에는 한 번만 값을 할당해야 한다는 제약이 생긴 것이다.

```
class CapturingLambdaExample {

    private int portNumber = 8080;

    public static void main(String[] args) {
        new CapturingLambdaExample().run();
    }

    public void run() {
        Runnable r = () -> System.out.println(portNumber);
        portNumber = 8443;
        r.run(); // 8443 출력
    }
}
```

그래서 람다가 메소드 바디에 있는 변수가 아닌 클래스의 멤버 변수를 참조하도록 코드를 작성하면 값을 변경할 수 있다.

```
class CapturingLambdaExample {

    public static void main(String[] args) {
        createLambda().run();
    }

    public static Runnable createLambda() {
        int portNumber = 8080;
        return () -> System.out.println(portNumber);
    }
}
```

아래와 같이 함수 구현체를 반환 받아 실행하는 것도 가능하다.

```
class CapturingLambdaExample{
    public static void main(String[] args) {
        SubCapturingLambdaExample subCapturingLambdaExample = new SubCapturingLambdaExample();
        Runnable runnable  = subCapturingLambdaExample.getLambda();
        subCapturingLambdaExample.portNumber = 8443;
        runnable.run();
    }

}
class SubCapturingLambdaExample {
    public int portNumber = 8080;

    public Runnable getLambda() {
        return () -> System.out.println(portNumber);
    }
}
```

위 코드의 경우 8443이 콘솔에 출력되는데, Runnable을 반환 받더라도 참조하는 대상이 SubCapturingLambdaExample의 portNumber이기 때문이다.

Predicate 조합

Predicate는 `and` 와 `or` 메소드로 조합함으로써 더 복잡한 로직을 만드는 것이 가능하다.
두 메소드 모두 디폴트 메소드로 구현되었으며, IntPredicate와 같은 기본형 함수형 인터페이스는 지원하지 않는다.

또한 타입 파라미터가 다른 경우에는 조합이 불가하다.

```
class Service {
    public int portNumber;
    public String serviceName;

    public Service(int portNumber, String serviceName) {
        this.portNumber = portNumber;
        this.serviceName = serviceName;
    }
}

class CapturingLambdaExample{
    public static void main(String[] args) {
        Service jordyBlogService = new Service(8443, "죠르디 블로그");

        Predicate<Service> testPortNumber = service -> service.portNumber == 8443;
        Predicate<Service> testServiceName = service -> service.serviceName.equals("죠르디 블로그");
        Predicate<Service> testJordyService = testPortNumber.and(testServiceName);

        System.out.println(testJordyService.test(jordyBlogService));
    }
}
```

Function 조합

Function도 `compose`라는 메소드와 `andThen`이라는 메소드를 사용해 조합하는 것이 가능하다.

`compose`는 파라미터로 넣어주는 함수를 먼저 실행하지만, `andThen`은 해당 함수 구현체를 먼저 실행한다.

두 메소드는 디폴트 메소드로 구현되었으며, 기본형 함수형 인터페이스는 지원하지 않는다.
또한 타입 파라미터가 다른 경우에는 조합이 불가하다.

```
class FunctionComposition{
    public static void main(String[] args) {
        Function<Integer, Integer> plusOne = x -> x + 1;
        Function<Integer, Integer> multiplyThree = x -> x * 3;

        Function<Integer, Integer> firstMultiplyThreeSecondPlusOne
            = plusOne.compose(multiplyThree);
        Function<Integer, Integer> firstPlusOneSecondMultiplyThree
            = plusOne.andThen(multiplyThree);

        System.out.println(firstMultiplyThreeSecondPlusOne.apply(3)); // 10
        System.out.println(firstPlusOneSecondMultiplyThree.apply(2)); // 9
    }
}
```