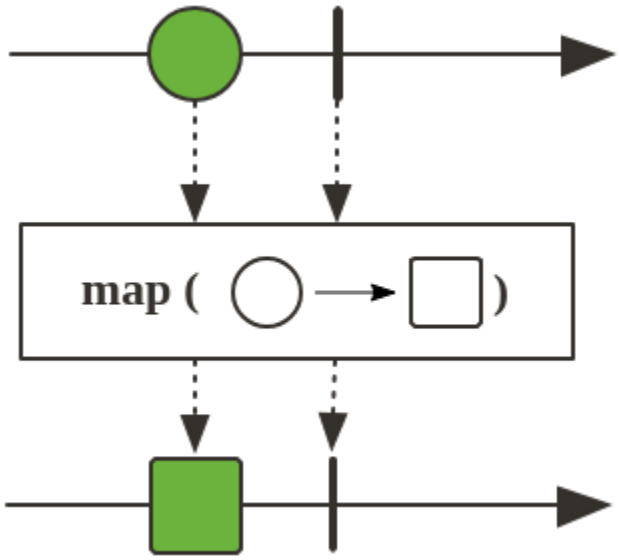
Reactor - map, flatMap

1 개요

블로킹/논블로킹, 동기/비동기 개념을 이해하고 있는 개발자를 대상으로, Reactor에서 map, flatMap 연산자가 어떤 특성을 가지고 있는지 설명하고, 개발 업무 수행 시 연산자 선택 기준을 제공한다.

2 Mono

2.1 map



> *Transform the item emitted by this `Mono` by applying a synchronous function to it. 이 `Mono` 로부터 방출된 아이템에 동기 함수를 적용하여 바꾼다.*

눈여겨 봐야 하는 부분은 동기 함수라는 표현이다. 여기서 동기 함수는 입력을 받은 값을 다른 값으로 변환하는 등의 CPU 연산만으로 해결 가능한 기능을 말한다. 즉, 네트워크 요청, 파일 입출력 등의 I/O 작업은 포함되지 않는다.

```
@Override
public void onNext(T t) {
        if (done) {
                Operators.onNextDropped(t, actual.currentContext());
                return;
        }

        R v;

        try {
                v = mapper.apply(t);
                if (v == null) {
                        throw new NullPointerException("The mapper [" + mapper.getClass().getName() + "]
returned a null value.");
                }
        }
        catch (Throwable e) {
                Throwable e_ = Operators.onNextError(t, e, actual.currentContext(), s);
                if (e_ != null) {
                        onError(e_);
                }
                else {
                        s.request(1);
                }
                return;
        }

        actual.onNext(v);
}
```

위의 코드는 Mono#map이 의존하는 MonoMap, MonoMap이 의존하는 FluxMap의 구현의 일부이다. 이를 통해 알 수 있는 점은 아래와 같다.

- 방출된 아이템을 처리 중인 스레드가 연속해서 처리한다.
- mapper의 결과가 null이면 NPE(NullPointerException)이 발생한다.

동기적으로 동작하는 블로킹 I/O 또한 동기 함수에 포함시켜 빌드하고 동작하게 할 수 있으나, I/O가 끝날 때까지 커널이 제어권을 돌려주지 않기 때문에 현재 발행자를 구독하고 있는 스레드가 다른 작업을 할 수 있는 기회를 잃는다.

이와 같은 경우는 비동기적으로 동작하는 논블로킹 I/O를 먼저 고려해 보고, 피치 못하게 동기 블로킹 I/O를 사용해야 하는 경우는 아래처럼 감싸 활용하도록 공식 문서에서 가이드하고 있다.

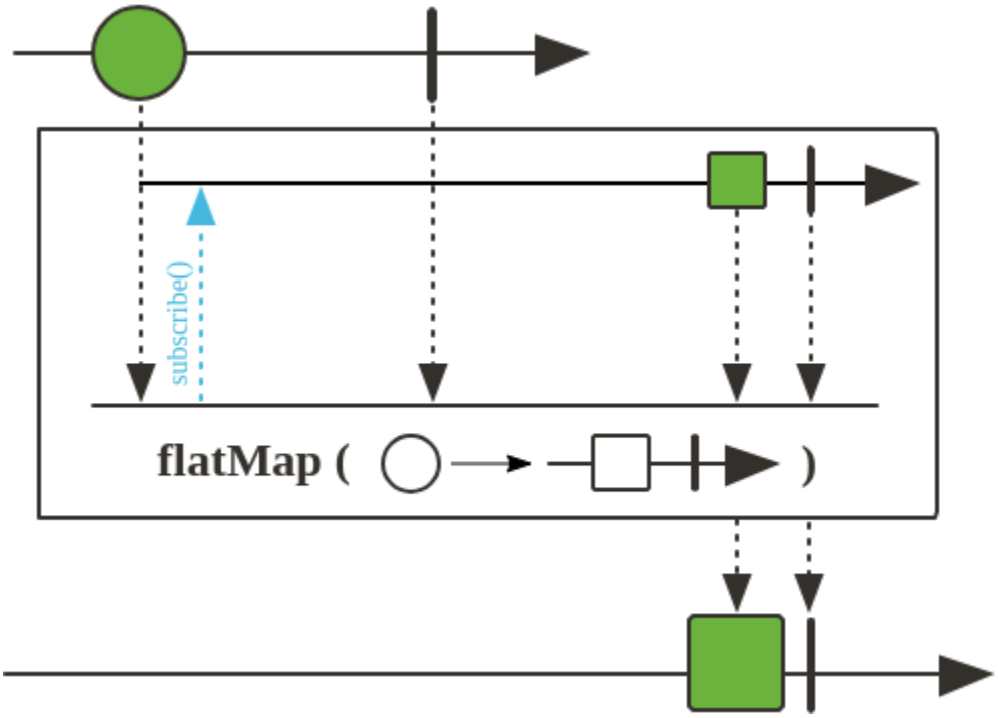
```
Mono.fromCallable(() -> {
    return /* make a remote synchronous call */
}).subscribeOn(Schedulers.boundedElastic());
```

참고로 `Schedulers.boundedElastic()`은 블로킹 리소스가 논블로킹 프로세스에 영향을 주지 않게 하도록 전용 스레드를 유지하고 관리하는 용도로 설계된 스케줄러이다.

2.1.1 예제

```
public interface CenterOfGravityReactiveService {
    Mono<CenterOfGravity> retrieveById(LoadControlId loadControlId);
    // ...
}

class SomeReactiveService {
    private final CenterOfGravityReactiveService centerOfGravityReactiveService;

    // constructor

    public Mono<CenterOfGravityResponse> retrieveById(final LoadControlId id) {
        return centerOfGravityReactiveService
                .retrieveById(id)
                .map(SomeReactiveService::toCenterOfGravityResponse);
    }

    private static CenterOfGravityResponse toCenterOfGravityResponse(final CenterOfGravity cog) {
        return /* ... */;
    }
}
```

2.2 flatMap



> *Transform the item emitted by this `Mono` asynchronously, returning the value emitted by another `Mono` (possibly changing the value type). 이 `Mono` 로부터 비동기적으로 발행된 아이템을 다른 `Mono`로 발행된 값으로 바꾼다. (이때 값의 타입이 바뀔 수 있다.*

핵심적인 표현은 발행된 아이템을 다른 Mono로 발행된 값으로 바꾼다는 점이다. 다른 Mono는 별도의 발행자이기 때문에 즉시 실행이 가능한지 알 수 없다. 따라서 이는 비동기적 동작을 처리하기 위해 설계되었음을 알 수 있다.

```
@Override
public void onNext(T t) {
    if (done) {
        Operators.onNextDropped(t, actual.currentContext());
        return;
    }
    done = true;

    Mono<? extends R> m;

    try {
        m = Objects.requireNonNull(mapper.apply(t),
                "The mapper returned a null Mono");
    }
    catch (Throwable ex) {
        actual.onError(Operators.onOperatorError(s, ex, t,
                actual.currentContext()));
        return;
    }

    if (m instanceof Callable) {
        @SuppressWarnings("unchecked") Callable<R> c = (Callable<R>) m;

        R v;
        try {
            v = c.call();
        }
        catch (Throwable ex) {
            actual.onError(Operators.onOperatorError(s, ex, t,
                    actual.currentContext()));
            return;
        }

        if (v == null) {
            actual.onComplete();
        }
        else {
            actual.onNext(v);
            actual.onComplete();
        }
        return;
    }

    try {
        m.subscribe(new FlatMapInner<>(this));
    }
    catch (Throwable e) {
        actual.onError(Operators.onOperatorError(this, e, t,
                actual.currentContext()));
    }
}
```

위는 Mono#flatMap이 내부적으로 의존하는 MonoFlatMap 클래스 구현의 일부이다. 이를 통해 알 수 있는 점은 아래와 같다.

- mapper를 통해 실행된 결과가 null이면 안 된다.
    - 적어도 Mono.empty()여야 한다.
- mapper를 통해 생성된 Mono가 다시 구독된다.
    - 실행하는 스레드가 스케줄러에 의해 바뀔 수 있다.

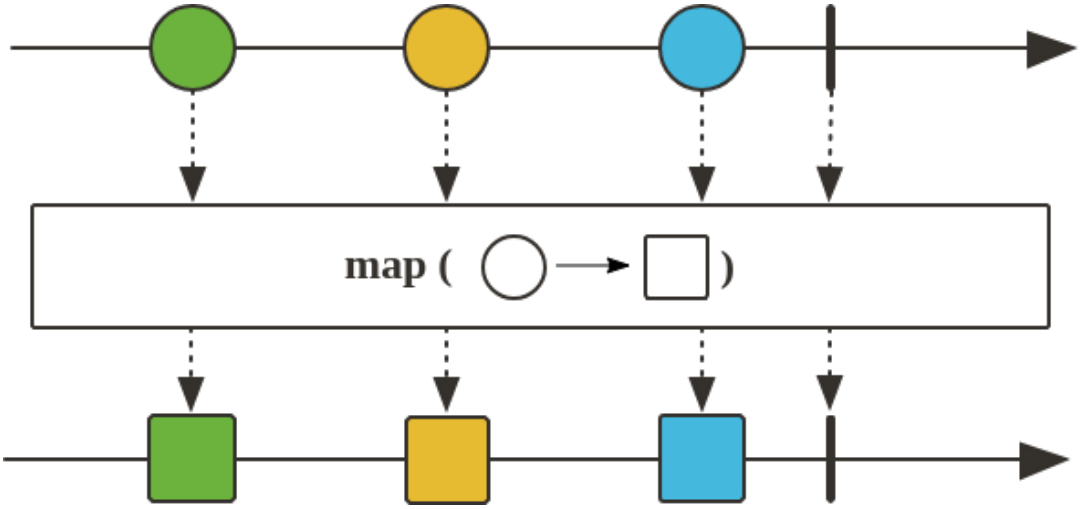
또한 Mono는 방출하는 아이템이 없거나 하나일 수 있기 때문에, 발행된 값을 다시 처리하는 Mono가 아이템을 방출하지 않으면 최종적으로 아무런 아이템도 발행되지 않고 종료할 수 있다.

2.2.1 예제

```
interface SessionReactiveService {
    Mono<Session> retrieve(Token token);
}

interface UserReactiveService {
    Mono<User> retrieve(UserId id);
}

class SomeReactiveService {
    private final SessionReactiveService sessionReactiveService;
    private final UserReactiveService userReactiveService;

    // constructor

    public Mono<User> retrieveCurrentUser(final Token token) {
        return sessionReactiveService.retrieve(token)
                .flatMap(userReactiveService::retrieve);
    }
}
```

3 Flux

3.1 map



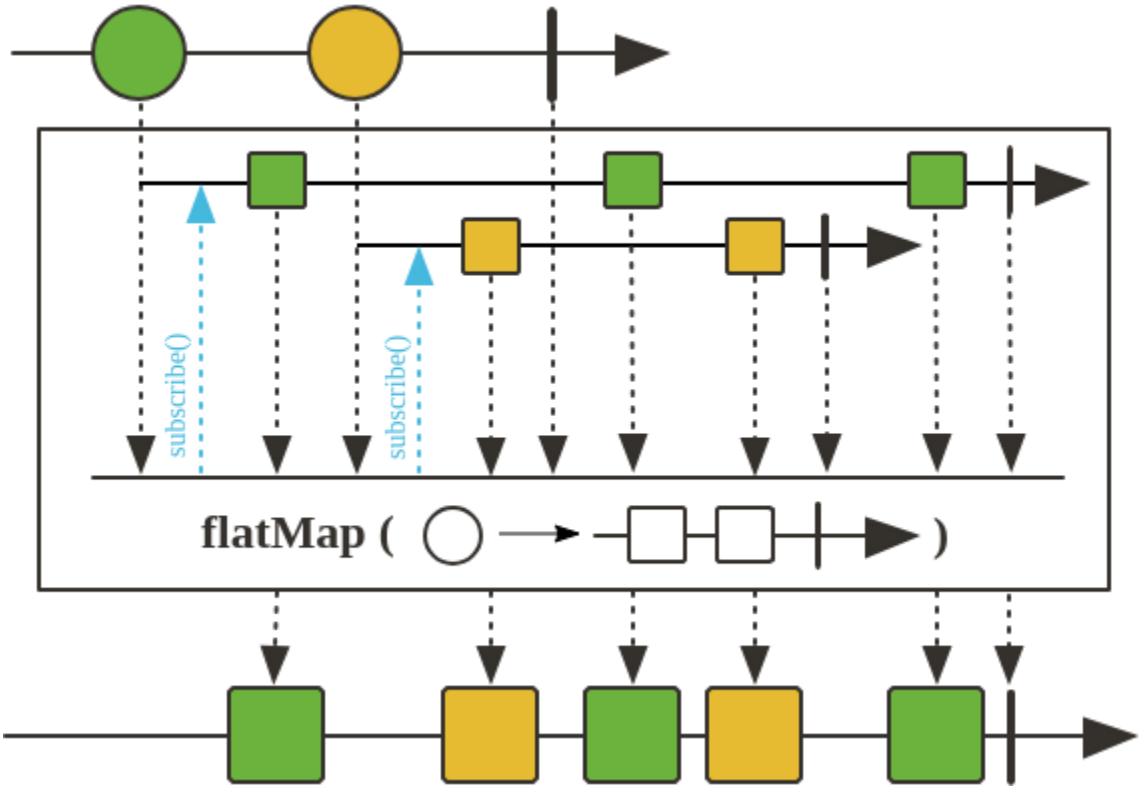
> *Transform the items emitted by this* ***Flux*** *by applying a synchronous function to each item.* ***Flux****가 방출한 각 아이템들에 동기 함수를 적용한다.*

Mono#map이 결국 FluxMap에 의존하는 것을 보면 알 수 있듯이, 최초의 발행자와 최종 발행자가 Flux이기 때문에 Mono와 달리 다건의 아이템을 발행할 수 있다는 점만 다르고, 다른 특성은 Mono#map과 동일하다.

3.1.1 예제

```
interface UserReactiveService {
    Flux<User> retrieveAll(DeparmentId deptId);
}

class SomeReactiveService {
    private final UserReactiveService userReactiveService;

    // constructor

    public Mono<User> retrieveAllUserName(final DeparmentId deptId) {
        return userReactiveService.retrieveAll(token)
                .map(User::name);
    }
}
```

3.2 flatMap



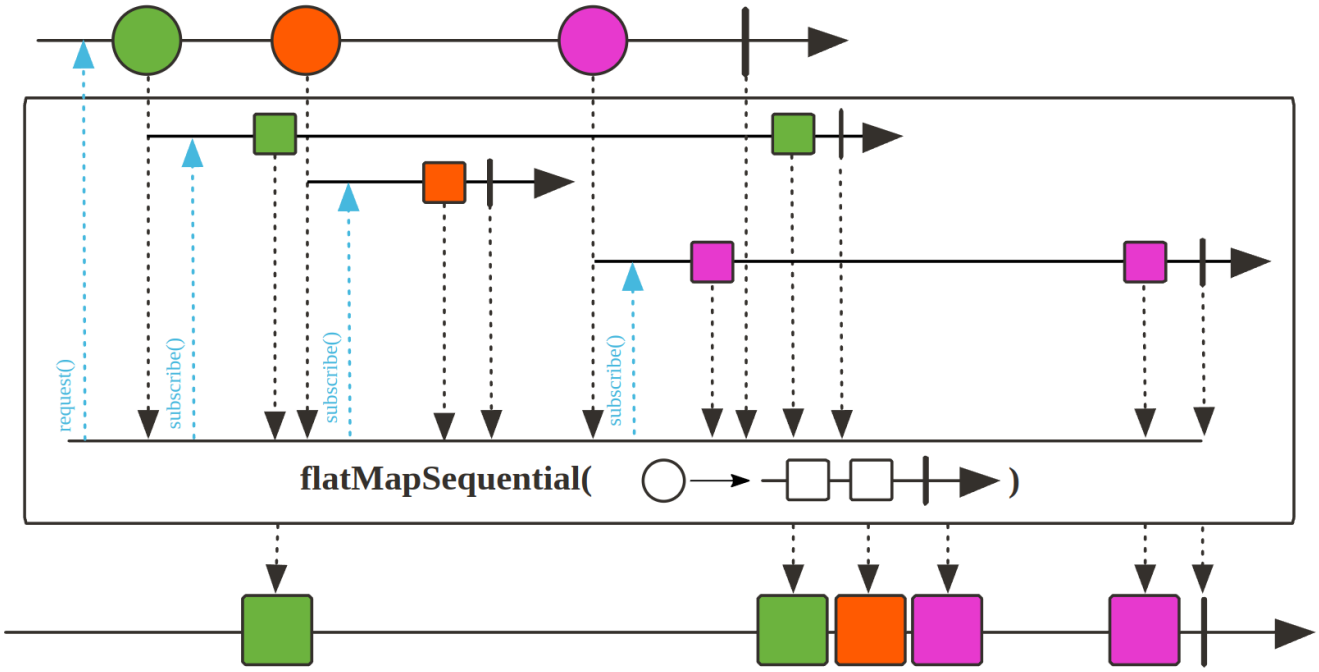
> *Transform the elements emitted by this `Flux` asynchronously into Publishers, then flatten these inner publishers into a single `Flux` through merging, which allow them to interleave. Flux에서 비동기적으로 발행한 각 요소를 발행자로 변환한 다음, 이러한 내부 발행자들을 하나의 Flux로 평탄화(flatten)하는 것을 허용한다. 이때 내부 발행자들을 통해 발행된 요소는 교차(interleave)할 수 있다.*

Mono#flatMap과 유사하지만, 최초 발행자로부터 파생되는 발행자가 하나가 아니기 때문에 차이가 발생한다. Flux#flatMapSequential과 concatMap과 비교하여 이해하면 조금 더 수월하다.

Flux#flatMap

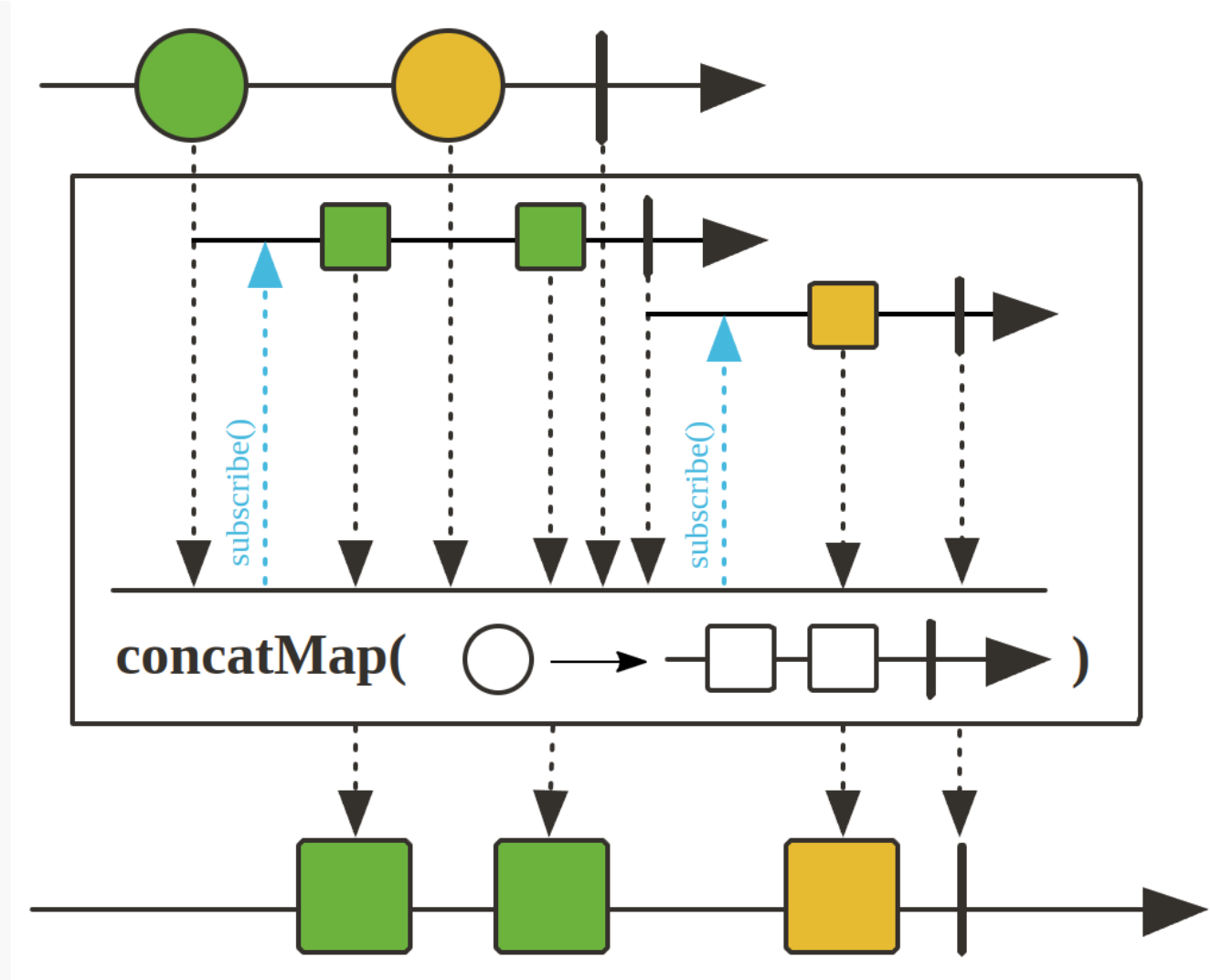
- 내부 발행자는 생성 즉시 구독된다.
- 내부 발행자에서 발행된 요소들은 내부 발행자가 생성된 순서와 관계 없이 최종 발행자를 통해 방출된다.

Flux#flatMapSequential



- 내부 발행자는 생성 즉시 구독된다.
- 내부 발행자에서 발행된 요소들은 내부 발행자가 생성된 순서대로 최종 발행자를 통해 방출된다.
- 기존에 방출된 요소를 잠시 담아둬야 하기 때문에 Flux#flatMap보다 메모리를 더 사용한다.

Flux#concatMap



- 내부 발행자는 순서대로 구독되며, 앞선 발행자가 종료되었을 때 다음 내부 발행자의 구독이 이루어진다.
- 구독 자체가 순차적으로 이루어지기 때문에
    - 내부 발행자가 생성된 순서대로 최종 발행자를 통해 방출된다.
    - 병렬성이 낮다.

3.2.1 예제

```
public class FluxFlatMapMain {
    public static void main(final String[] args) {
        Flux.range(1, 5)
                .flatMap(SampleMain::asyncOperation)
                .subscribe(System.out::println); // 1 5     .

        try {
            Thread.sleep(1100); // flatMap       1   .
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    private static Mono<String> asyncOperation(final int value) {
        return Mono.delay(Duration.ofSeconds(1)) // 1
                .map(delay -> value + " processed");
    }
}
```

```
public class FluxFlatMapSequentialMain {
    public static void main(final String[] args) {
        Flux.range(1, 5)
                .flatMapSequential(SampleMain::asyncOperation)
                .subscribe(System.out::println); // 1 5  .

        try {
            Thread.sleep(1100); // flatMapSequential       1  .
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    private static Mono<String> asyncOperation(final int value) {
        return Mono.delay(Duration.ofSeconds(1)) // 1
                .map(delay -> value + " processed");
    }
}
```

```
public class FluxConcatMapMain {
    public static void main(final String[] args) {
        Flux.range(1, 5)
                .concatMap(SampleMain::asyncOperation)
                .subscribe(System.out::println); // 1 5  .

        try {
            Thread.sleep(6100); // concatMap       6  .
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    private static Mono<String> asyncOperation(final int value) {
        return Mono.delay(Duration.ofSeconds(1)) // 1
                .map(delay -> value + " processed");
    }
}
```

4 요약

- Reactor를 이용한 프로그래밍에서 I/O는 비동기적으로 처리해야 한다.
    - 동기 블로킹 I/O는 Mono#fromCallable 등의 메소드를 통해 감싸서(wrap) 논블로킹 프로세싱에 영향을 주면 안 된다.
- map, flatMap, concatMap 등에 입력 인자로 활용되는 mapper 함수는 null을 반환하면 안 된다.
    - 발행할 수 있는 요소가 없다면 Mono.empty() 또는 Flux.empty()를 반환해야 한다.
- mapper 함수가 동기적으로 동작하는 게 바람직하다면, Mono#map 또는 Flux#map을 활용한다.
- mapper 함수가 비동기적으로 동작하는 게 바람직하다면,
    - Mono의 경우 Mono#flatMap을 사용한다.
    - Flux의 경우
        - 내부 발행자의 순서대로 정렬이 불필요하면
            - Flux#flatMap을 사용한다.
        - 내부 발행자의 순서대로 정렬이 필요하면서
            - 병렬성이 필요하다면 Flux#flatMapSequential을 사용한다.
            - 병렬성이 필요 없다면 Flux#concatMap을 사용한다.

5 추가 자료

- 블로킹/논블로킹, 동기/비동기
- Reactor 요약
- reactor-extra 시작하기