Stream operator characteristics

sanghyuck.na@lge.com

```
IntStream newIntStream() { return IntStream.rangeClosed(0, 100);}
Stream<Integer> newObjStream() {  return Stream.iterate(0, i -> i
+ 1).limit(100); }
```

1

Boxing

- Primitive type값이 Wrapper class로 바뀌는 동작
- Autoboxing
  - JVM Compiler에 의해 Boxing이 되는 현상

Primitive type		Wrapper class
boolean	Boxing →	Boolean
byte		Byte
char		Character
float		Float
int		Integer
long		Long
short	← Unboxing	Short
double		Double

```
char priv = 'c';

Character obj = null;

priv = obj;
```

```
List<Integer> list =;
for (int i = 1; i < 10; ++i)
    list.add(i);
```

2

Primitive Stream

- Stream Autoboxing
  - 실행 속도개선의 최우선 항목은 Autoboxing으로 인한 성능저하로 그 차이는 약 11배 입니다. 스트림의 중간연산자 수가 증가하고 unboxing도 동시에 존재하는 경우 최악의 성능은 O(n x n) 이상에 달합니다.
- Primitive Stream
  - autoboxing, unboxing을 피하기 위한 primitive type데이터처리를 위한 스트림
  - IntStream, LongStream, DoubleStream

```
Stream.iterate(1, i -> i + 1).limit(Integer.MAX_VALUE)
    .forEach((i) -> {});

IntStream.iterate(1, i -> i + 1).limit(Integer.MAX_VALUE)
    .forEach((i) -> {});
```

```
auto_boxing      avgt    40 42362710.284 ± 381518.247 us/op
no_boxing        avgt    40 3651694.469 ± 104729.861 us/op

                         42362710.284 / 3651694.469 = 11.6배
```

3

Special API for Primitive type

- Number: iterate(), rangeClosed()
- Boxing: boxed(), mapTo(Int|Long|Double)()
- Array: toArray()

```
Stream<T> iterate(T seed, UnaryOperator<T> f)
IntStream iterate(int seed, IntUnaryOperator f)

static IntStream rangeClosed(int startInclusive,
    int endInclusive)
```

```
Stream<Integer> os = newIntStream().boxed();
IntStream ps = newObjStream().mapToInt(Integer::valueOf);

DoubleStream ps2 = newIntStream().asDoubleStream();

Integer[] arrs = newObjStream().toArray(Integer[]::new);
int[] arrp = newIntStream().toArray();
```

4

특별한 메소드

- Primitive stream 통계함수
  - sum(), average(), max(), min()
- 공통 함수
  - count()
  - 순서유지 반복함수 forEachOrdered()

```
int sum = newIntStream().sum();
OptionalDouble avg = newIntStream().average();
OptionalInt max = newIntStream().max();
OptionalInt min = newIntStream().min();
```

```
long cnt = newIntStream().count();
newObjStream().forEach(System.out::println);
newObjStream().forEachOrdered(System.out::println);
```

5

State

- Stateless operator
  - 새로운 데이터처리 시 이전에 데이터를 반영해온 상태를 유지하지 않는 연산자
- Stateful operator
  - 현재 결과를 계산하기 위해서는 과거 입력 값을 유지해야 하는 연산자

```
filter, map, flotmap
Stream<T> concat(Stream<? extends T> a, Stream<? extends T> b)
Stream<T> peek(Consumer<? super T> action)
Stream<T> limit(long maxSize)
Stream<T> skip(long n)
```

```
Stream<T> distinct()
Stream<T> sorted()
Stream<T> sorted(Comparator<? super T> comparator)
```

6

Stateless

- concate()
  - 첫 번째 스트림의 모든 요소 다음에 두 번째 스트림의 모든 요소를 결합한 느슨한결합스트림(lazily-coupling stream) 작성
  - Input 둘 중에 하나라도 parallel이라면 Parallel
- peek()
  - 전달하는 action으로 실행 중인 결과 값을 입력으로 볼 수 있는 Stream작성

```
<T> Stream<T> concat(Stream a, Stream b)
Stream<T> peek(Consumer<? super T> action)
```

```
Stream<Integer> ls = Stream.of(1, 2, 3);
Stream<Integer> rs = Stream.of(4, 5, 6);
Stream<Integer> s = Stream.concat(ls, rs);

List<String> l = Stream.of(1, 2, 3)
    .peek(System.out::println)
    .collect(Collectors.toList());
```

7

Stateful

- distinct() 중복제거
- sorted() 정렬

```
Stream<T> distinct()
Stream<T> sorted()
Stream<T> sorted(Comparator<T> comparator)
```

```
newIntStream().distinct();

newIntStream().sorted();
newIntStream().sorted(Comparator.reverseOrder());
```

8

Short-circuit

- 일부 데이터로 평가를 완료하는 연산자
  - limit(), skip()

```
Stream<T> limit(long maxSize)

Stream<Integer> s = newIntStream().limit(50);
```

```
Stream<T> skip(long n)

Stream<Integer> s = newIntStream().skip(20);
```

```
Optional<T> findAny()

Optional<Integer> s = newIntStream().findAny();
```

9

Deterministic

- Deterministic operator
  - 동일한 데이터를 다른 시간 다시 입력해도 결과가 항상 같은 연산자
  - findFist(), forEachOrdered(), iterate(), `limit()`, `skip()`
- Non-deterministic operator
  - 항상 다른 결과가 나타나는 연산자
  - 데이터순서보다 처리시간 우선 처리하여 멀티스레드에 유용
  - findAny(), forEach(), `generate()`

```
newObjStream().findFirst();
newObjStream().forEachOrdered(System.out::println);

newObjStream().parallel().findAny();
newObjStream().parallel().forEach(System.out::println);
```

10

스트림 생성전략

- 처리목적 특성
  - 순차 순서: 높은 자원활용을 위한 배치작업에 적합
  - 병렬 비순서: 빠른 처리속도가 필요한 데이터산출 및 집계작업에 적합
  - 병렬 순서: 빠른 응답속도가 필요한 UI 이벤트처리작업에 적합
- 자료구조 특성
  - 검색속도 위주 처리: Hash, Set(≠ Array, LinkedList)
  - 인덱싱속도 위주 처리: Array(≠LinkedList)

Scheduling	Sequential sequential()	Parallel parallel() parallelStream();
Ordering	Order <Default>	Unorder unorder()

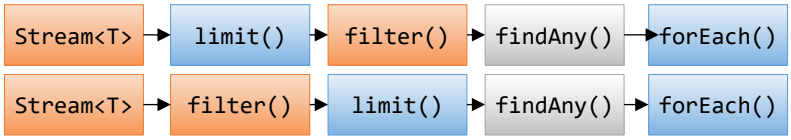
```
IntStream.of(1, 2, 3)
    .sequential()
    .sum();
IntStream.of(1, 2, 3)
    .parallel().unordered()
    .sum();
```

```
IntStream.of(1, 2, 3)
    .parallel()
    .sum();
```

11

연산자 사용전략

- Stateless operation
  - 연산속도와 메모리사용량 최소화, stateful인 경우unordered()호출
- Short-circuit
  - 되도록 전방배치
- Non-deterministic operation
  - 순서유지속성이 없는 연산자 선택
- 연산자 배치
  - 전방에 데이터수가 비교적 많이 감소되는 연산자선택



12

distinct, unordered

- distinct()
  - Parallel pipeline에서 상당한 memory barrier로 지연이 상당히 발생하여 순서유지가 필요 없다면 되도록 unordered()를 호출해서 실행속도 개선
- 18991559.435 37133629.644 = 0.511, 50%성능 개선

```
Random r = new Random();
ArrayList<Integer> mData = r.ints().limit(100000).collect(ArrayList::new,
ArrayList::add, ArrayList::addAll);

List<Integer> l = mData.parallelStream().distinct()
     .collect(Collectors.toList());
List<Integer> l = mData.parallelStream().unordered().distinct()
     .collect(Collectors.toList());
```

```
Result "co.ajava.lib.jmh.MainBenchmark.nodistinct":
18991559.435 ±(99.9%) 1063311.893 ns/op [Average]
  (min, avg, max) = (16084327.494, 18991559.435, 23712771.124), stdev =
1419490.955 CI (99.9%): [17928247.542, 20054871.327] (assumes normal
distribution)

Benchmark                  Mode  Cnt         Score         Error  Units
MainBenchmark.distinct     avgt   25  37133629.644 ± 1194895.379  ns/op
MainBenchmark.nodistinct   avgt   25  18991559.435 ± 1063311.893  ns/op
```

13

Multithread degree

- core수 조회와 설정 API

```
Runtime.getRuntime().availableProcessors();
ForkJoinPool.commonPool().getParallelism();
System.getProperty(
    "java.util.concurrent.ForkJoinPool.common.parallelism");
```

```
Executors.newFixedThreadPool(5);
System.setProperty(
    "java.util.concurrent.ForkJoinPool.common.parallelism", "5");
java -Djava.util.concurrent.ForkJoinPool.common.parallelism=5
```

14

도전하세요!

- 잘못 된 원인은 무엇이고 총 몇 개인가요?
- 찾아본 것을 적용해보세요!

```
요구사항: 빠른처리속도
ManagementFactory.getMemoryPoolMXBeans().stream()
    .map(MemoryPoolMXBean::getUsage)
    .map(MemoryUsage::getUsed)
    .reduce(0L, Long::sum);
```

```
요구사항: 빠른응답속도
ManagementFactory.getMemoryPoolMXBeans().stream()
    .flatMap(b -> Arrays
    .stream(b.getMemoryManagerNames()))
    .distinct().forEach(System.out::println);
```

15

정리

- 성능요소 고려사항
  - Boxing
  - 특별한 메소드
  - 스트림 생성전략
  - 연산자 배치
  - State
  - Short-circuit
  - Deterministic
  - distinct + unordred
  - Multithread degree

16

Lambda Exception Handling

sanghyuck.na@ge.com

Function type input parameter API
Unchecked exception handler in Lambda

17

Function type input parameter API

- Post code runs failed
  - f.run()에서 Exception이 발생하면 s.run()은 호출되지 못합니다.
  - f.run()의 실행여부에 상관없이 s.run()은 실행되도록 합니다.

```
doInOrder(
    () -> { throw new IllegalStateException();},
    () -> System.out.println("second run")
);

void doInOrder(Runnable f, Runnable s) {
    f.run();
    s.run();
}
```

18

Forward post code

- Forward exception check failed
  - f.run()에서 Exception이 발생하면 적절한 종료코드 s.run()을 실행하지 못합니다.
  - s.run()는 f.run()의 Exception 사실을 알지 못합니다.
  - f.run()가 오류를 발생하면 handler를 추가합니다.

```
doInOrder(
    () -> { throw new IllegalStateException();},
    () -> System.out.println("second run")
);

void doInOrder(Runnable f, Runnable s) {
    new Thread(()-> {
        f.run();
    }).start();
    s.run();
}
```

19

Exception Handler

- Forward return loss
  - f.run()의 리턴값이나 오류발생을 s.run()가 알아야할 필요가 있습니다.
  - 먼저 s.run()의 입력매개변수로 f.run()의 리턴값을 전달합니다.

```
doInOrder(
    () -> {throw new IllegalStateException();},
    () -> System.out.println("second run"),
    System.out::println
);

void doInOrder(Runnable f, Runnable s,
                   Consumer<Throwable> handler) {
    try {
        f.run();
    } catch (Throwable t) {
        handler.accept(t);
    }
    s.run();
}
```

20

Bypass forward return value

- Forward exception loss
  - f.get()가 Exception을 발생하면 s.accept()는 호출되지 않습니다.
  - s.run()은 항상 f.get()의 리턴값을 전달하고, Exception이 발생하면 null을 전달합니다.

```
doInOrder(() -> 0,
    (r) -> System.out.println("second run="),
    System.out::println);

<T> void doInOrder(Supplier<T> f, Consumer<T> s,
                   Consumer<Throwable> handler) {
    try {
        T result = f.get();
        s.accept(result);
    } catch (Throwable t) {
        handler.accept(t);
    }
}
```

21

Final code

- Function type input parameter API
  - Post code runs failed, Forward exception check failed
  - Forward return loss, Forward exception loss
  - Bypass forward return value

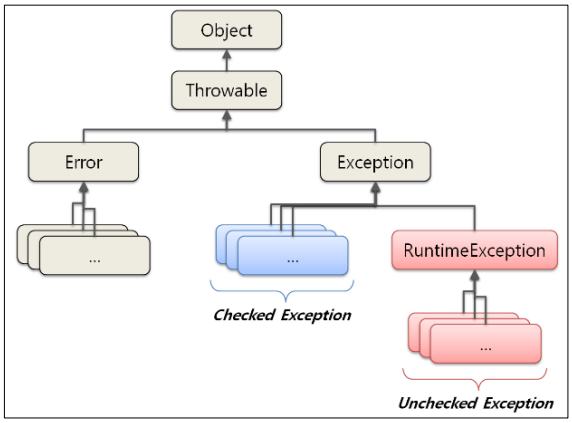
```
doInOrder4(
    () -> {return Integer.valueOf(0);},
    (first, e) -> {
        System.out.println(first);
        if (!Objects.isNull(e))
            e.printStackTrace(); });

<T> void doInOrder4(Supplier<T> first, BiConsumer<T, Throwable> second)
{
    T result = null;
    try { result = first.get();
        second.accept(result, null);
    } catch (Throwable t) {
        second.accept(result, t);
    }
}
```

22

Unchecked exception handler in Lambda

- Unchecked exception
  - Compile오류가 발생하지 않기 때문에 오류처리를 하지않는 경우가 많으며, 그로 인해 예상치 못한 오류와 미정의 동작이 발생합니다. Lambda expression은 Runtime시 출력된 로그상에 오류가 발생한 인스턴스 명은 JVM이 임의로 부여한 객체이름으로 추적이 쉽지도 않습니다.



23

JAVA 프로그래머 Exception 처리 준수사항

	Checked Exception	Unchecked Exception
처리의무	O	X
확인시점	Compile time	Runtime
후행조건	(없음)	Rollback (원상복귀)
예	IOException SQLException	NullPointException IllegalArgumentException IndexOutOfBoundException SystemException

24

Handling checked exception in Lambda expression

- Files.readAllBytes() IOException
- throws
  - Method throws 선언 추가는 interface 규약과 맞지 않아 Compile 오류입니다

```
final Path P = Paths.get("setupact.log");
Supplier<String> s = () -> {
    return new String(Files.readAllBytes(P),
        StandardCharsets.UTF_8);
};
```

```
final Path P = Paths.get("setupact.log");
Supplier<String> s = new Supplier<String>() {
    public String get() throws IOException {
        return new String(Files.readAllBytes(P),
                          StandardCharsets.UTF_8);
    }
};
```

25

Hidden typhoon

- Try-catch
  - 단, Lambda 외부에서는 오류발생유무를 식별할 방법이 없습니다. 또는 throw new로 Exception propagation다면 문제는 원점으로 되돌아갑니다.

```
final Path P = Paths.get("setupact.log")
Supplier<String> s = () -> {
    try {
        return new String(Files.readAllBytes(P),
                          StandardCharsets.UTF_8);
    } catch(IOException e) {
        throw e;
    }
};
```

26

Throw new

- Unchecked exception
  - Interface method 선언을 그대로 유지합니다.
  - 호출자(Caller)는 unchecked exception을 처리할 의무가 있습니다.
  - 외부에서 Exception유무확인과 후처리 코드를 추가할 수 있습니다.

```
final Path P = Paths.get("setupact.log");
Supplier<String> s = () -> {
    try {
        return new String(Files.readAllBytes(P),
                          StandardCharsets.UTF_8);
    } catch(IOException e) {
        throw new RuntimeException(e);
    }
};
```

27

Normalization

- 포괄적인 Handling

```
final Path P = Paths.get("setupact.log");
Supplier<String> s = () -> {
    try {
        return new String(Files.readAllBytes(P),
                          StandardCharsets.UTF_8);
    } catch(Exception e) {
        throw new RuntimeException(e);
    }
};
```

```
@FunctionalInterface
public interface Callable<V> {
    V call() throws Exception;
}
```

28

Final

```
<T> Supplier<T> unchecked(Callable<T> c) {
     return () -> {
        try {
            return c.call();
        }catch(Exception e) {
            throw new RuntimeException(e);
        }
    };
}
```

```
final Path P = Paths.get("setupact.log");
Supplier<String> s1 = unchecked(
    () -> new String(Files.readAllBytes(P),
                     StandardCharsets.UTF_8));

Supplier<Void> s2 = unchecked(() -> null);
```

29

정리

- Function type input parameter API
- Unchecked exception handler in Lambda

30