

7장. 병렬 데이터 처리와 성능

숫자 n 을 인수로 받아서 1부터 n 까지의 모든 숫자의 합계를 반환하는 메서드 구현

전통적인 자바 방식



```
1 public long iterativeSum(long n) {  
2     long result = 0;  
3     for (long i = 1L; i ≤ n; i++) {  
4         result += i;  
5     }  
6     return result;  
7 }
```

n 이 커진다면 이 연산을 병렬로 처리하는 것이 좋다.(하지만)

- 결과 변수는 어떻게 동기화해야 할까?
- 몇 개의 스레드를 사용해야 할까?
- 숫자는 어떻게 생성할까?
- 생성된 숫자는 누가 더할까?

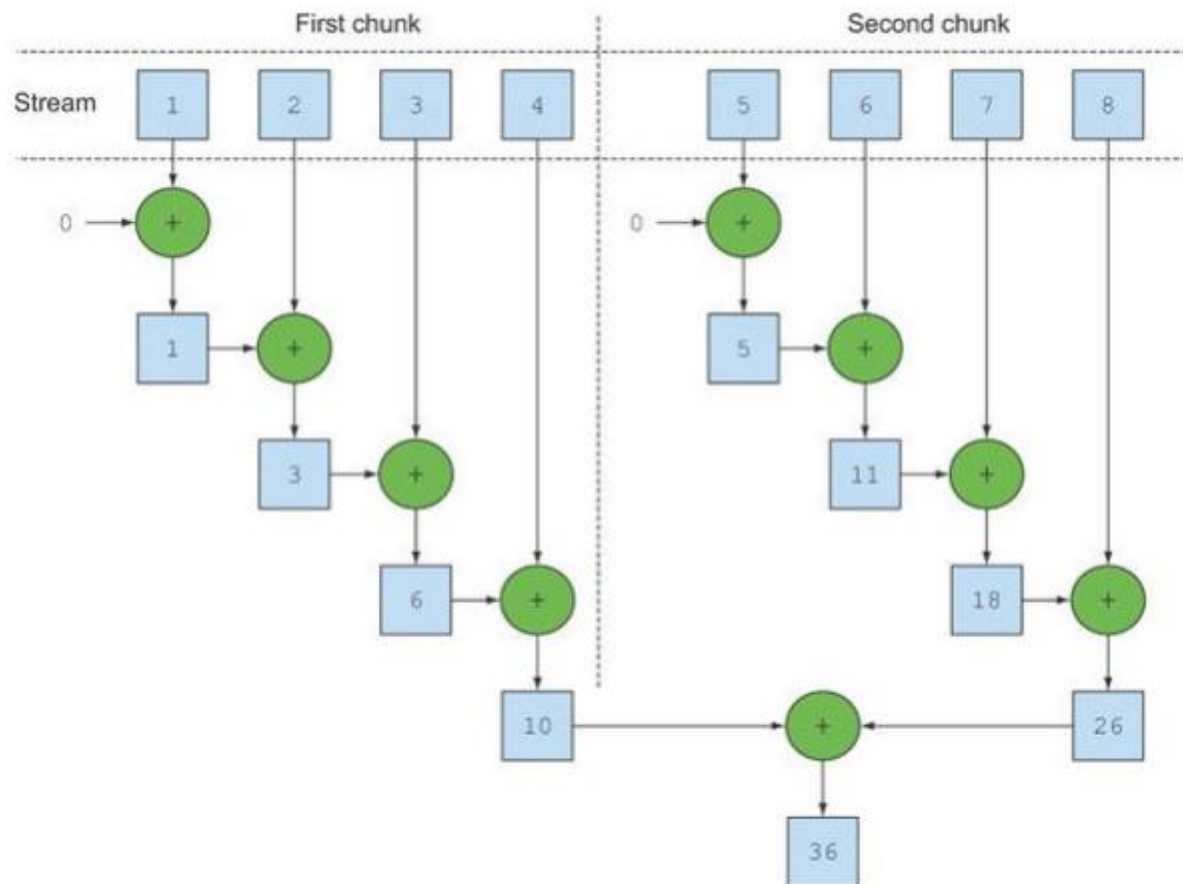
다음과 같은 문제점을 고려해야 한다.

순차 스트림에 `parallel` 메서드를 호출하면 기존의 함수형 리듀싱 연산 (숫자 합계 계산)이 병렬로 처리된다.

숫자 `n`을 인수로 받아서 1부터 `n`까지의 모든 숫자의 합계를 반환하는 메서드 구현



```
1 public long sequentialSum(long n) {  
2     return Stream.iterate(1L, i -> i + 1) // 무한 자연수 스트림 생성  
3         .limit(n) // n개 이하로 제한  
4         .parallel() // 스트림을 병렬 스트림으로 변환  
5         .reduce(0L, Long::sum); // 모든 숫자를 더하는 스트림 리듀싱 연산  
6 }
```



하나의 연산을 세 가지 (반복형, 순차 리듀싱, 병렬 리듀싱) 중 어느 것이 가장 빠를까?

자바 마이크로벤치마크 하니스(Java Microbenchmark Harness)

- 어노테이션 기반 방식 지원
- 안정적으로 자바 프로그램이나 자바 가상 머신(JVM)을 대상으로 하는 다른 언어용 벤치마크 구현 가능
- Maven, Gradle 빌드 도구 등 JMH 의존성 추가해 사용가능

하나의 연산을 세 가지 (반복형, 순차 리듀싱, 병렬 리듀싱) 중 어느 것이 가장 빠를까?

Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz

순차 리듀싱

```
1 @BenchmarkMode(Mode.AverageTime) // 벤치마크 대상 메서드를 실행하는 데 걸린 평균 시간 측정
2 @OutputTimeUnit(TimeUnit.MILLISECONDS) // 벤치마크 결과를 밀리초 단위로 출력
3 @Fork(2, jvmArgs={"-Xms4G", "-Xmx4G"}) // 4Gb의 힙 공간을 제공한 환경에서 두 번
4 public Class ParallelStreamBenchmark { // 벤치마크를 수행해 결과의 신뢰성 확보
5     private static final long N= 10_000_000L;
6
7     @Benchmark // 벤치마크 대상 메서드
8     public long sequentialSum() {
9         return Stream.iterate(1L, i -> i + 1).limit(N)
10             .reduce(0L, Long::sum);
11     }
12
13     @TearDown(Level.Invocation) // 매 번 벤치마크를 실행한 다음에는 가비지 컬렉터 동작 시도
14     public void tearDown() {
15         System.gc();
16     }
17 }
18
```

Benchmark	Mode	Cnt	Score	Error	Units
ParallelStreamBenchmark.sequentialSum	avgt	4	76.394 ± 5.276		ms/op

하나의 연산을 세 가지 (반복형, 순차 리듀싱, 병렬 리듀싱) 중 어느 것이 가장 빠를까?

반복형

```
1 @BenchmarkMode(Mode.AverageTime) // 벤치마크 대상 메서드를 실행하는 데 걸린 평균 시간 측정
2 @OutputTimeUnit(TimeUnit.MILLISECONDS) // 벤치마크 결과를 밀리초 단위로 출력
3 @Fork(2, jvmArgs={"-Xms4G", "-Xmx4G"}) // 4Gb의 힙 공간을 제공한 환경에서 두 번
4 public Class ParallelStreamBenchmark { // 벤치마크를 수행해 결과의 신뢰성 확보
5     private static final long N= 10_000_000L;
6
7     @Benchmark
8     public long iterativeSum() {
9         long result = 0;
10        for (long i = 1L; i ≤ N; i++) {
11            result += i;
12        }
13        return result;
14    }
15
16    @TearDown(Level.Invocation) // 매 번 벤치마크를 실행한 다음에는 가비지 컬렉터 동작 시도
17    public void tearDown() {
18        System.gc();
19    }
20 }
21
```

기본값을 박싱하거나 언박싱할 필요가 없으므로 더 빠른 것으로 예상할 수 있다.

Benchmark	Mode	Cnt	Score	Error	Units
ParallelStreamBenchmark.iterativeSum	avgt	4	4.325	± 0.345	ms/op

순차 리듀싱보다 약19배 빠르다.

하나의 연산을 세 가지 (반복형, 순차 리듀싱, 병렬 리듀싱) 중 어느 것이 가장 빠를까?

병렬 리듀싱

```
1 @BenchmarkMode(Mode.AverageTime) // 벤치마크 대상 메서드를 실행하는 데 걸린 평균 시간 측정
2 @OutputTimeUnit(TimeUnit.MILLISECONDS) // 벤치마크 결과를 밀리초 단위로 출력
3 @Fork(2, jvmArgs={"-Xms4G", "-Xmx4G"}) // 4Gb의 힙 공간을 제공한 환경에서 두 번
4 public Class ParallelStreamBenchmark { // 벤치마크를 수행해 결과의 신뢰성 확보
5     private static final long N= 10_000_000L;
6
7     @Benchmark
8     public long parallelSum() {
9         return Stream.iterate(1L, i -> i + 1)
10             .limit(N)
11             .parallel()
12             .reduce(0L, Long::sum);
13     }
14
15     @TearDown(Level.Invocation) // 매 번 벤치마크를 실행한 다음에는 가비지 컬렉터 동작 시도
16     public void tearDown() {
17         System.gc();
18     }
19 }
20
```

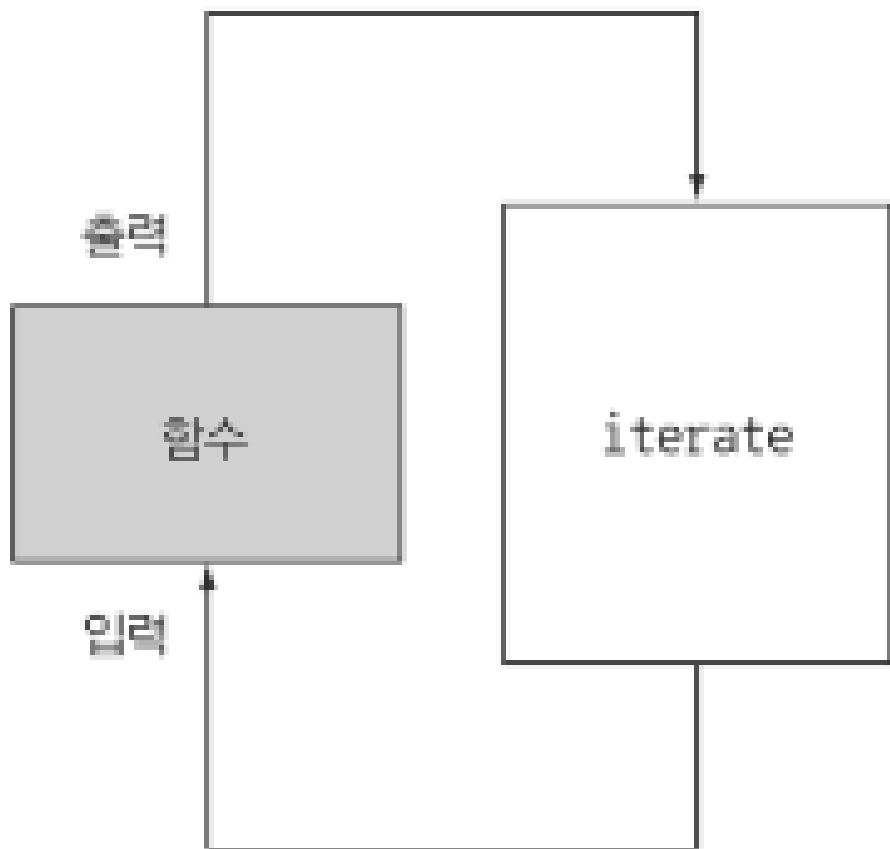
Benchmark	Mode	Cnt	Score	Error	Units
ParallelStreamBenchmark.parallelSum	avgt	4	98.416	± 8.061	ms/op

4core를 활용했지만 순차 리듀싱보다도 느리게 나왔다.

병렬 스트림의 두가지 문제점

- 반복 결과로 박싱된 객체가 만들어지므로 숫자를 더하려면 언박싱을 해야 한다.
- 반복 작업은 병렬로 수행할 수 있는 독립 단위로 나누기가 어렵다.

그림 7-2 `iterate`는 본질적으로 순차적이다.



- 이전 연산의 결과에 따라 다음 함수의 입력이 달라지기 때문에 `iterate` 연산을 청크로 분할하기가 어렵다.
- 리듀싱 연산이 수행되지 않는다. 리듀싱 과정을 시작하는 시점에 전체 숫자 리스트가 준비되지 않았으므로 스트림을 병렬로 처리할 수 있도록 청크로 분할할 수 없다.

결국 병렬로 처리한다고 지시를 해도 스레드를 할당하는 오버헤드만 증가하게 된다.

멀티코어 프로세서를 활용해서 효과적으로 함께 연산을 병렬로 실행하려면 어떻게 해야 할까?

LongStream.rangeClosed

- LongStream.rangeClosed는 기본형 long을 직접 사용하므로 박싱과 언박싱 오버헤드가 사라진다.
- LongStream.rangeClosed는 쉽게 청크로 분할할 수 있는 숫자 범위를 생산한다. 예를 들어 1-20 범위의 숫자를 각각 1-5, 6-10, 11-15, 16-20 범위의 숫자로 분할할 수 있다.

```
1 @Benchmark
2 public long rangedSum() {
3     return LongStream.rangeClosed(1, N).reduce(0L, Long::sum);
4 }
5
```

Benchmark	Mode	Cnt	Score	Error	Units
ParallelStreamBenchmark.rangedSum	avgt	4	5.863 ± 0.280		ms/op

```
1 @Benchmark
2 public long parallelRangedSum() {
3     return LongStream.rangeClosed(1, N).parallel().reduce(0L, Long::sum);
4 }
5
```

Benchmark	Mode	Cnt	Score	Error	Units
ParallelStreamBenchmark.rangedSum	avgt	4	5.863	± 0.280	ms/op

숫자 스트림 처리 속도가 더 빠르다. 특화되지 않은 스트림을 처리할 때는 오토박싱, 언박싱 등의 오버헤드를 수반하기 때문이다.

상황에 따라서는 어떤 알고리즘을 병렬화하는 것보다 적절한 자료구조를 선택하는 것이 더 중요하다는 사실을 단적으로 보여준다.

```
1 @Benchmark
2 public long parallelRangedSum() {
3     return LongStream.rangeClosed(1, N)
4         .parallel()
5         .reduce(0L, Long::sum);
6 }
```

Benchmark	Mode	Cnt	Score	Error	Units
ParallelStreamBenchmark.parallelRangedSum	avgt	4	1.498 ± 2.049		ms/op

원하는 결과가 나왔다.

Benchmark	Mode	Cnt	Score	Error	Units
ParallelStreamBenchmark.iterativeSum	avgt	4	4.325 ± 0.345		ms/op

하지만 멀티코어 간의 데이터 이동은 생각보다 비싸기 때문에 코어 간의 데이터 전송 시간보다 훨씬 오래 걸리는 작업만 병렬로 다른 코어에서 수행하는 것이 바람직하다.

병렬 스트림을 잘못 사용하면서 발생하는 많은 문제는 공유된 상태를 바꾸는 알고리즘을 사용하기 때문에 일어난다.

다음은 n 까지의 자연수를 더하면서 공유된 누적자를 바꾸는 프로그램을 구현한 코드이다.

```
1 public long sideEffectSum(long n) {  
2     Accumulator accumulator = new Accumulator();  
3     LongStream.rangeClosed(1, n).forEach(accumulator::add);  
4     return accumulator.total;  
5 }  
6  
7 public class Accumulator {  
8     public long total = 0;  
9     public void add(long value) { total += value; }  
10 }
```

```
1 public static long sideEffectParallelSum(long n) {  
2     Accumulator accumulator = new Accumulator();  
3     LongStream.rangeClosed(1, n).parallel().forEach(accumulator::add);  
4     return accumulator.total();  
5 }
```

```
Result: 13682010592045  
Result: 6813159025489  
Result: 7394442150042  
Result: 8129338165547  
Result: 8731862463259  
Result: 5928083095069  
Result: 4289237988668  
Result: 8911358704921  
Result: 8531943438118  
Result: 7149191970959  
SideEffect parallel sum done in: 1 msecs
```

올바른 결과값(50000005000000)이 나오지 않는다.

여러 스레드에서 공유하는 객체의 상태를 바꾸는 forEach 블록 내부에서 add 메서드를 호출하면서 이 같은 문제가 발생한다.

지금은 공유된 가변 상태를 피해야 한다는 사실만 기억하자!!

‘천 개 이상의 요소가 있을 때만 병렬 스트림을 사용하라’ 와 같이 양을 기준으로 병렬 스트림 사용을 결정하는 것은 적절하지 않다.

- 확신이 서지 않으면 직접 측정하라.
- 박싱을 주의하라. 기본형 특화 스트림(IntStream, LongStream, DoubleStream)을 이용하는 것이 좋다.
- 순차 스트림보다 병렬 스트림에서 성능이 떨어지는 연산이 있다. ex) limit, findFirst
- 스트림에서 수행하는 전체 파이프라인 연산 비용을 고려하라.
ex) 처리해야 할 요소 수가 N , 하나의 요소를 처리하는데 드는 비용 Q , 전체 처리 비용 $N*Q$
 Q 가 높아진다는 것은 병렬 스트림으로 성능을 개선할 수 있는 가능성이 있음
- 소량의 데이터에서는 병렬 스트림이 도움 되지 않는다.

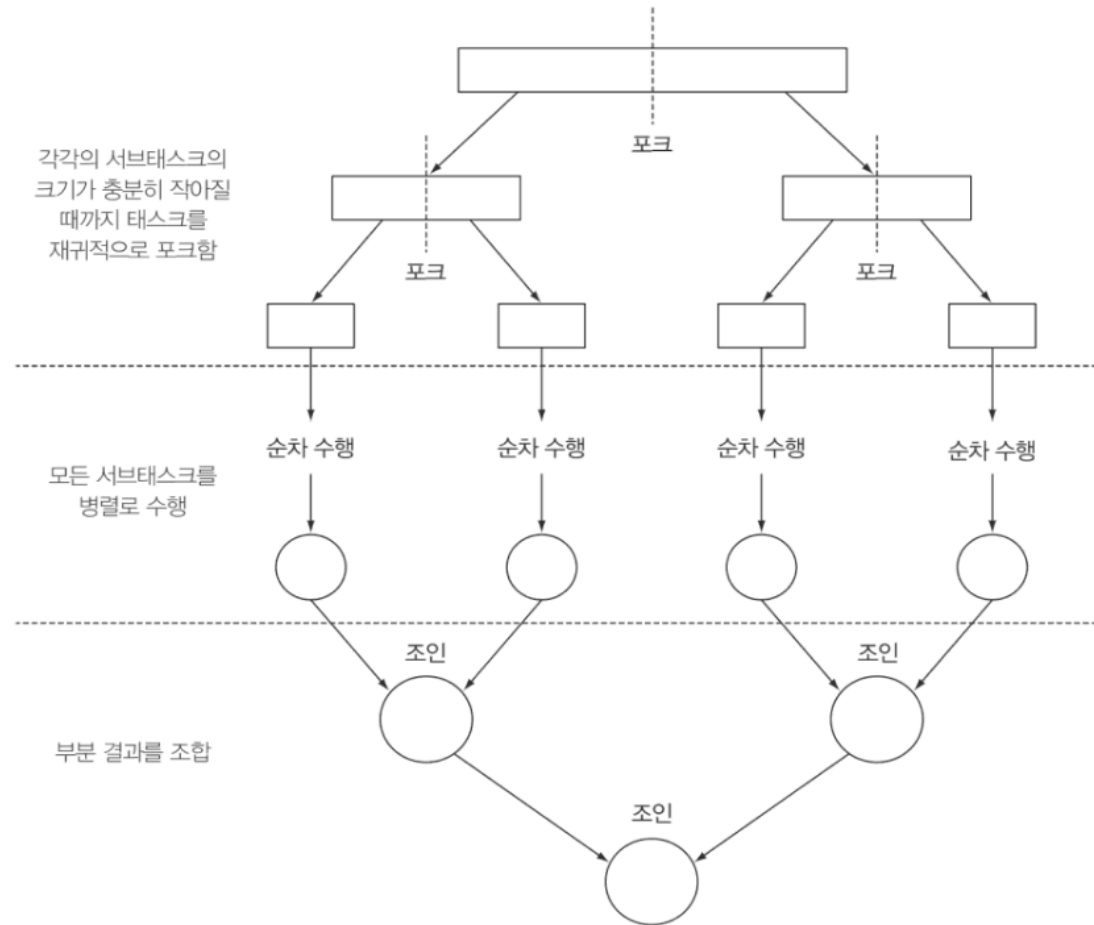
- 스트림을 구성하는 자료구조가 적절한지 확인하라 ex) `ArrayList` > `LinkedList`
- 스트림의 특성과 파이프라인의 중간 연산이 스트림의 특성을 어떻게 바꾸는지에 따라 분해 과정의 성능이 달라질 수 있다.
ex) `SIZED` 스트림은 정확히 같은 크기의 두 스트림으로 분할할 수 있으므로 효과적으로 스트림을 병렬 처리 할 수 있다.
필터 연산이 있으면 스트림의 길이를 예측할 수 없어 효과적으로 스트림을 병렬 처리할 수 있을지 알 수 없게 된다.
- 최종 연산의 병합 과정 비용을 살펴보라. ex) `Collector`의 `combiner` 메서드

표 7-1 스트림 소스와 분해성

소스	분해성
<code>ArrayList</code>	훌륭함
<code>LinkedList</code>	나쁨
<code>IntStream.range</code>	훌륭함
<code>Stream.iterate</code>	나쁨
<code>HashSet</code>	좋음
<code>TreeSet</code>	좋음


포크/조인 프레임워크는 병렬화할 수 있는 작업을 재귀적으로 작은 작업으로 분할한 다음에 서브태스크 각각의 결과를 합쳐서 전체 결과를 만들도록 설계되었다.

그림 7-3 포크/조인 과정



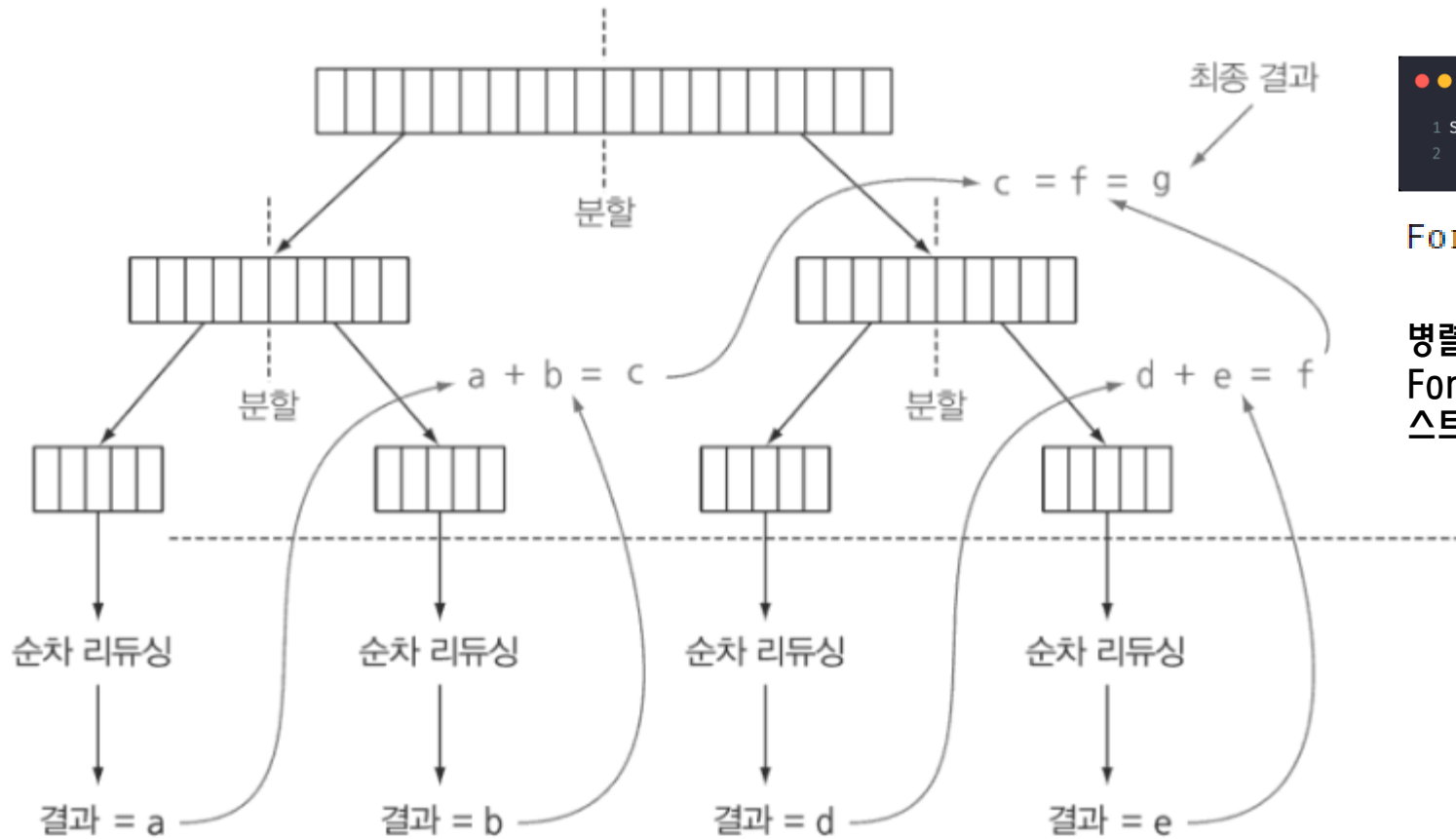
```
1 public class ForkJoinSumCalculator
2     extends java.util.concurrent.RecursiveTask<Long> { // 포크/조인 프레임워크에서 사용할 태스크를 생성
3     private final long[] numbers; // 더할 숫자 배열
4     private final int start; // 이 서브태스크에서 처리할 배열의 초기 위치
5     private final int end; // 와 최종 위치
6     private static final long THRESHOLD = 10_000; // 이 값 이하의 서브태스크는 더 이상 분할할 수 없다.
7
8     public ForkJoinSumCalculator(long[] numbers) { // 메인 태스크를 생성할 때 사용할 공개 생성자
9         this(numbers, 0, numbers.length);
10    }
11
12    private ForkJoinSumCalculator(long[] numbers, int start, int end) {
13        this.numbers = numbers; // 메인 태스크의 서브 태스크를 재귀적으로 만들 때
14        this.start = start; // 사용할 비공개 생성자
15        this.end = end;
16    }
```

```
1 @Override
2 protected Long compute() { //RecursiveTask의 추상 메서드 오버라이드
3     int length = end - start; // 이 태스크에서 더할 배열의 길이
4     if (length ≤ THRESHOLD) {
5         return computeSequentially(); // 기준값과 같거나 작으면 순차적으로 결과를 계산
6     }
7
8     ForkJoinSumCalculator leftTask =
9         new ForkJoinSumCalculator(numbers, start, start + length/2); // 배열의 첫 번째 절반을 더하도록 서브 태스크 생성
10    leftTask.fork(); // ForkJoinPool의 다른 스레드로 새로 생성한 태스크를 비동기로 실행
11    ForkJoinSumCalculator rightTask =
12        new ForkJoinSumCalculator(numbers, start + length/2, end); // 배열의 나머지 절반을 더하도록 서브태스크 생성
13    Long rightResult = rightTask.compute(); // 두 번째 서브태스크를 동기 실행 이때 추가로 분할이 일어날 수 있음
14    Long leftResult = leftTask.join(); // 첫 번째 서브태스크의 결과를 읽거나 아직 결과가 없으면 기다림
15    return leftResult + rightResult; // 두 서브태스크의 결과를 조합한 값이 이 태스크의 결과이다.
16 }
17
18 private long computeSequentially() { // 더 분할할 수 없을때 서브태스크의 결과를 계산하는 알고리즘
19     long sum = 0;
20     for (int i = start; i < end: i++) {
21         sum += numbers[i];
22     }
23     return sum;
24 }
25 }
```



```
1 public static long forkJoinSum(long n) {  
2     long[] numbers = LongStream.rangeClosed(1, n).toArray();  
3     ForkJoinTask<Long> task = new ForkJoinSumCalculator(numbers);  
4     return FORK_JOIN_POOL.invoke(task);  
5 }
```

그림 7-4 포크/조인 알고리즘



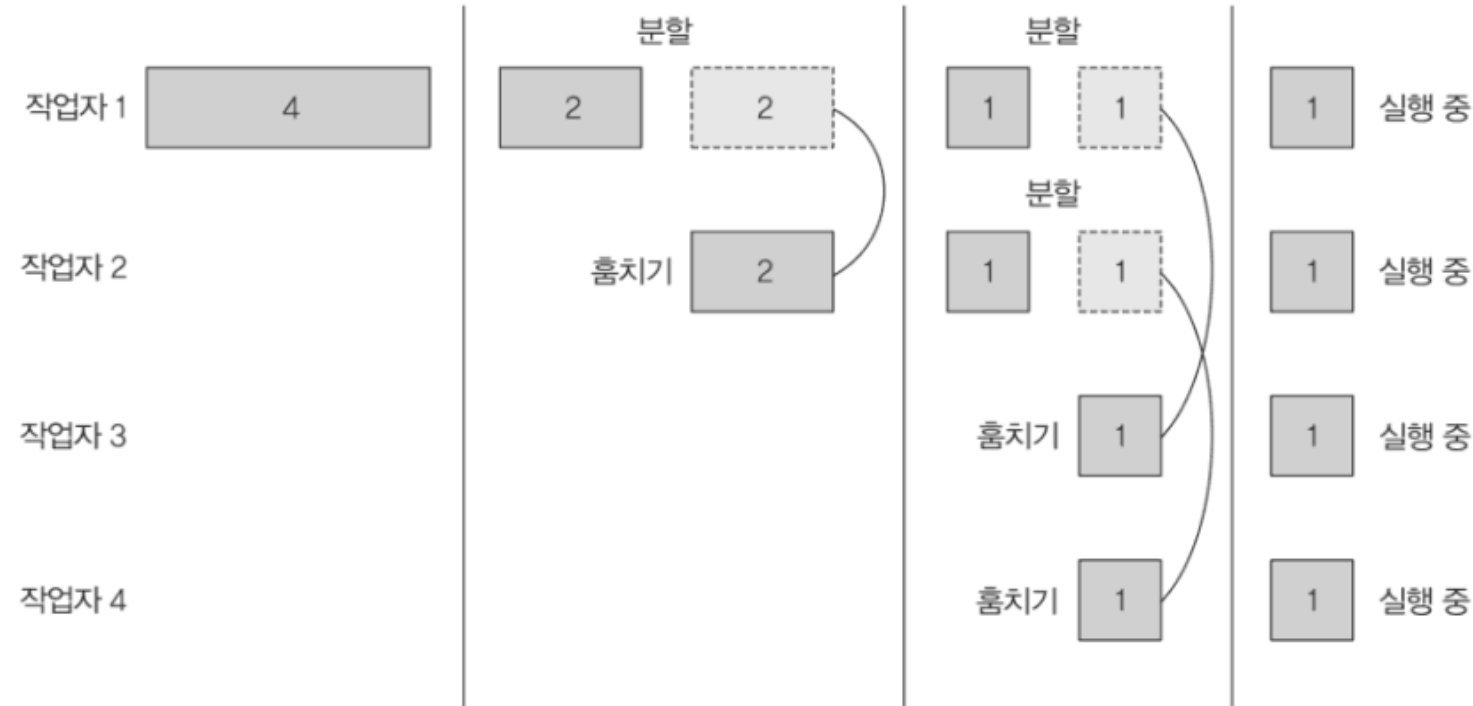
```
1 System.out.println("ForkJoin sum done in: " +
2     measurePerf(ForkJoinSumCalculator::forkJoinSum, 10_000_000L) + "msecs");
```

ForkJoin sum done in: 25msecs

병렬 스트림을 이용할 때보다 성능이 나빠졌다. 하지만 이는 ForkJoinSumCalculator 태스크에서 사용할 수 있도록 전체 스트림을 long[]으로 변환했기 때문이다.

- join 메서드를 태스크에 호출하면 태스크가 생산하는 결과를 준비될 때까지 호출자를 **블록시킨다**. 따라서 두 서브태스크가 모두 시작된 다음 join을 호출해야 한다.
- RecursiveTask 내에서는 ForkJoinPool의 **invoke** 메서드를 사용하지 말아야 한다. 대신 **compute**나 **fork** 메서드를 직접 호출할 수 있다. 순차 코드에서 병렬 계산을 시작할 때만 **invoke**를 사용한다.
- 서브태스크에 fork 메서드를 호출해서 ForkJoinPool의 일정을 조절할 수 있다. 왼쪽 작업과 오른쪽 작업 모두에 fork 메서드를 호출하는 것이 자연스러울 것 같지만 한 쪽 작업에는 **fork를 호출하는 것보다 compute를 호출하는 것이 효율적이다**. 그러면 두 서브 태스크의 한 태스크에는 같은 스레드를 재사용할 수 있으므로 풀에서 불필요한 태스크를 할당하는 **오버헤드**를 피할 수 있다.
- 포크/조인 프레임워크를 이용하는 병렬 계산은 디버깅 하기 어렵다. 보통 IDE로 디버깅할 때 스택 트레이스로 문제가 일어난 가정을 쉽게 확인할 수 있는데, 포크/조인 프레임워크에서는 fork라 불리는 다른 스레드에서 compute를 호출하므로 스택 트레이스가 도움이 되지 않는다.

그림 7-5 포크/조인 프레임워크에서 사용하는 작업 훑치기 알고리즘



- 각각의 스레드는 자신에게 할당된 태스크를 포함하는 이중 연결 리스트를 참조하면서 작업이 끝날 때 마다 큐의 헤드에서 다른 태스크를 가져와서 작업을 처리한다.
- 만약 한 스레드가 작업을 마쳤는데 더 이상 처리할 태스크가 없으면 다른 스레드 큐 꼬리에서 작업을 훑쳐온다. 모든 태스크가 작업을 끝낼 때 까지, 즉 모든 큐가 빌 때까지 이 과정을 반복한다.

Spliterator는 ‘분할할 수 있는 반복자(splitable iterator)’ 라는 의미다.



```
1 public interface Spliterator<T> {  
2     boolean tryAdvance(Consumer<? super T> action);  
3     Spliterator<T> strySplit();  
4     long estimateSize();  
5     int characteristics();  
6 }
```

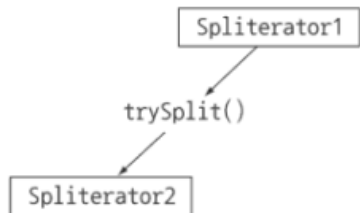
Spliterator는 ‘분할할 수 있는 반복자(splitable iterator)’ 라는 의미다.



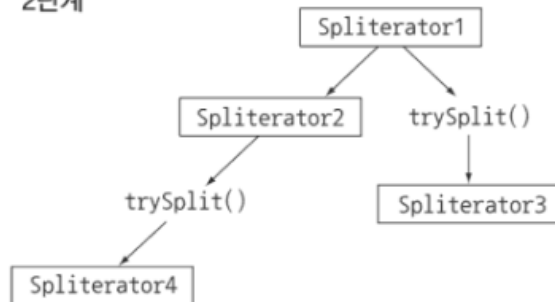
```
1 public interface Spliterator<T> {  
2     boolean tryAdvance(Consumer<? super T> action);  
3     Spliterator<T> strySplit();  
4     long estimateSize();  
5     int characteristics();  
6 }
```

그림 7-6 재귀 분할 과정

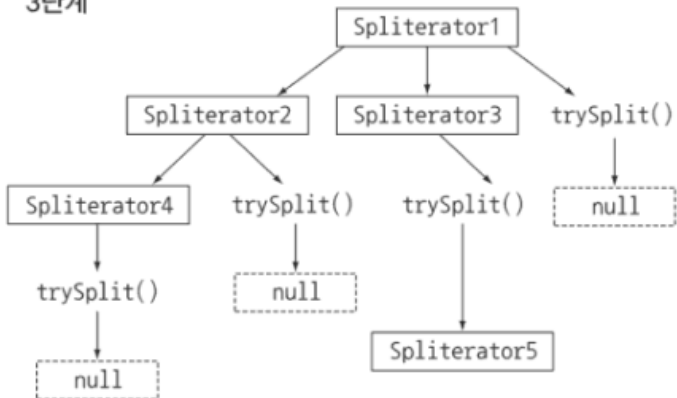
1단계



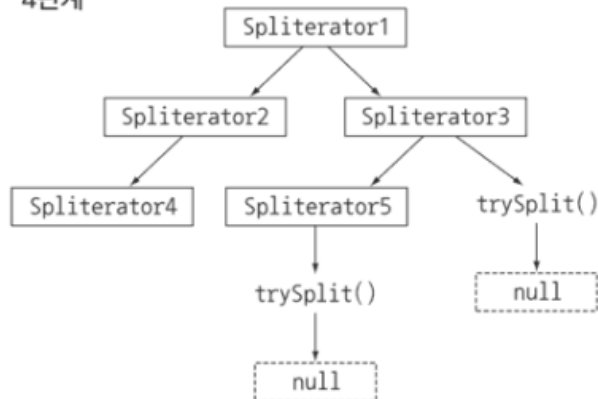
2단계



3단계



4단계



- 1단계: 첫 번째 Spliterator에 trySplit을 호출하면 두 번째 Spliterator가 생성된다.
- 2단계: 두 번째 Spliterator에 trySplit을 다시 호출하면 네 개의 Spliterator가 생성된다. trySplit의 결과가 null이 될 때까지 이 과정을 반복한다.
- 3단계: trySplit이 null을 반환했다는 것은 더 이상 자료구조를 분할할 수 없음을 의미한다.
- 4단계: Spliterator에 호출한 모든 trySplit의 결과가 null이면 재귀 분할 과정이 종료된다.

Characteristics 메서드는 Spliterator 자체의 특정 집합을 포함하는 int를 반환한다.

표 7-2 Spliterator 특성

특성	의미
ORDERED	리스트처럼 요소에 정해진 순서가 있으므로 Spliterator는 요소를 탐색하고 분할할 때 이 순서에 유의해야 한다.
DISTINCT	x, y 두 요소를 방문했을 때 x.equals(y)는 항상 false를 반환한다.
SORTED	탐색된 요소는 미리 정의된 정렬 순서를 따른다.
SIZED	크기가 알려진 소스(예를 들면 Set)로 Spliterator를 생성했으므로 estimatedSize()는 정확한 값을 반환한다.
NON-NULL	탐색하는 모든 요소는 null이 아니다.
IMMUTABLE	이 Spliterator의 소스는 불변이다. 즉, 요소를 탐색하는 동안 요소를 추가하거나, 삭제하거나, 고칠 수 없다.
CONCURRENT	동기화 없이 Spliterator의 소스를 여러 스레드에서 동시에 고칠 수 있다.
SUBSIZED	이 Spliterator 그리고 분할되는 모든 Spliterator는 SIZED 특성을 갖는다.

```
1 public int countWordsIteratively(String s) {
2     int counter = 0;
3     boolean lastSpace = true;
4     for (char c : s.toCharArray()) { // 문자열의 모든 문자를 하나씩 탐색한다.
5         if (Character.isWhitespace(c)) {
6             lastSpace = true;
7         } else {
8             if (lastSpace) counter++; // 문자를 하나씩 탐색하다 공백 문자를
9                                     // 만나면 지금까지 탐색한 문자를 단어로
10        }                               // 간주하여(공백 문자는 제외) 단어 수를
11    return counter;                   // 증가시킨다.
12 }
13
```

```
1 public static final String SENTENCE =  
2     " Nel mezzo del cammin di nostra vita "  
3     + "mi ritrovai in una selva oscura"  
4     + " che la dritta via era smarrita ";  
5  
6 public static void main(String[] args) {  
7     System.out.println("Found " + countWordsIteratively(SENTENCE) + " words");  
8 }
```

```
"C:\Program Files\java\jdk-11+28\bin\java.exe" ...
```

```
Found 19 words
```

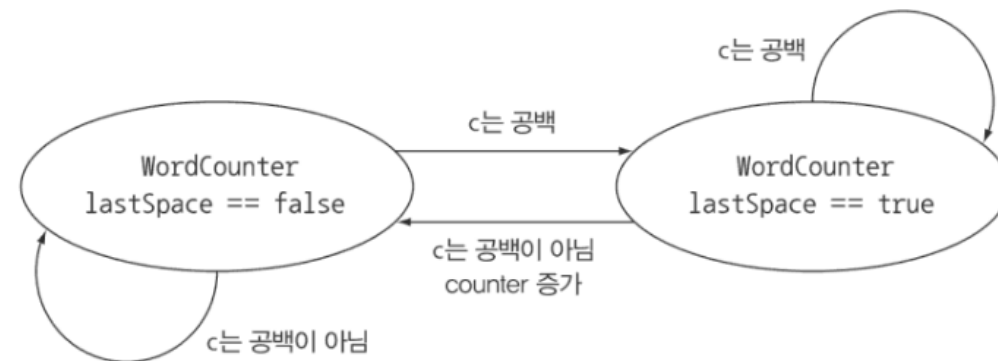
함수형으로 단어 수를 세는 메서드 재구현하기

```

1 private static class WordCounter {
2
3     private final int counter;
4     private final boolean lastSpace;
5
6     public WordCounter(int counter, boolean lastSpace) {
7         this.counter = counter;
8         this.lastSpace = lastSpace;
9     }
10
11     public WordCounter accumulate(Character c) { // 반복 알고리즘처럼 accumulate 메서드는
12         if (Character.isWhitespace(c)) { // 문자열의 문자를 하나씩 탐색한다.
13             return lastSpace ? this : new WordCounter(counter, true);
14         }
15         else { // 문자를 하나씩 탐색하다 공백 문자를
16             return lastSpace ? new WordCounter(counter + 1, false) : this; // 만나면 지금까지 탐색한 문자를 단어로 간주하여
17         } // (공백 문자는 제외) 단어 수를 증가시킨다.
18     }
19
20     public WordCounter combine(WordCounter wordCounter) {
21         return new WordCounter(counter + wordCounter.counter, // 두 WordCounter의 counter 값을 더한다.
22                                 wordCounter.lastSpace); // counter 값만 더할 것이므로 마지막 공백은 신경 쓰지 않는다.
23     }
24
25     public int getCounter() {
26         return counter;
27     }
28
29 }

```

그림 7-7 새로운 문자 c를 탐색했을 때 WordCounter의 상태 변화



```

1 private int countWords(Stream<Character> stream) {
2     WordCounter wordCounter = stream.reduce(new WordCounter(0, true),
3                                             WordCounter::accumulate,
4                                             WordCounter::combine);
5     return wordCounter.getCounter();
6 }
7
8 Stream<Character> stream = IntStream.range(0, SENTENCE.length())
9     .mapToObj(SENTENCE::charAt);
10 System.out.println("Found " + countwords(stream) + " words");

```




```
1 System.out.println("Found " + countWords(stream) + " words");  
2  
3 System.out.println("Found " + countWords(stream.parallel()) + " words");
```

```
"C:\Program Files\java\jdk-11+28\bin\java.exe" ...
```

```
Found 19 words countWords(stream)
```

```
Found 39 words countWords(stream.parallel())
```

WordCounter 병렬로 수행하기



```
1 private static class WordCounterSpliterator implements Spliterator<Character> {
2
3     private final String string;
4     private int currentChar = 0;
5
6     private WordCounterSpliterator(String string) {
7         this.string = string;
8     }
9
10    @Override
11    public boolean tryAdvance(Consumer<? super Character> action) {
12        action.accept(string.charAt(currentChar++)); // 현재 문자를 소비한다.
13        return currentChar < string.length(); // 소비할 문자가 남아있으면 true를 반환한다.
14    }
15
16    @Override
17    public Spliterator<Character> trySplit() {
18        int currentSize = string.length() - currentChar;
19        if (currentSize < 10) { // 파싱할 문자열을 순차 처리할 수 있을 만큼
20            return null; // 충분히 작아졌음을 알리는 null을 반환한다.
21        }
22        for (int splitPos = currentSize / 2 + currentChar;
23             splitPos < string.length(); splitPos++) { // 파싱할 문자열의 중간을 분할 위치로 설정한다.
24            if (Character.isWhitespace(string.charAt(splitPos))) { // 다음 공백이 나올 때까지 분할 위치를 뒤로 이동시킨다.
25                Spliterator<Character> spliterator = // 처음부터 분할 위치까지 문자열을 파싱할 새로운 WordCounterSpliterator를
26                    new WordCounterSpliterator(string.substring(currentChar, splitPos)); //생성한다.
27                currentChar = splitPos; // 이 WordCounterSpliterator의 시작 위치를 분할 위치로 설정한다.
28                return spliterator; // 공백을 찾았고 문자열을 분리했으므로 루프를 종료한다.
29            }
30        }
31        return null;
32    }
33
34    @Override
35    public long estimateSize() {
36        return string.length() - currentChar;
37    }
38
39    @Override
40    public int characteristics() {
41        return ORDERED + SIZED + SUBSIZED + NONNULL + IMMUTABLE;
42    }
43
44 }
```

Consumer는 스트림을 탐색하면서 적용해야 하는 함수 집합이 작업을 처리할 수 있도록 소비한 문자를 전달하는 자바 내부 클래스다.

예제에서는 스트림을 탐색하면서 하나의 리듀싱 함수, 즉 WordCounter의 accumulate 메서드만 적용한다.

탐색해야 할 요소의 개수(estimatedSize)는 Spliterator가 파싱할 문자열 전체 길이 (string.length())와 현재 반복 중인 위치(currentChar)의 차다.

마지막으로 characteristic 메서드는 프레임워크에 Spliterator가 ORDERED (문자열의 문자 등장 순서가 유의미함), SIZE (estimatedSize 메서드의 반환값이 정확함), SUBSIZED (trySplit으로 생성된 Spliterator도 정확한 크기를 가짐), NONNULL (문자열에는 null 문자가 존재하지 않음), IMMUTABLE (문자열 자체가 불변 클래스이므로 문자열을 파싱하면서 속성이 추가되지 않음) 등의 특성임을 알려준다.

- 내부 반복을 이용하면 명시적으로 다른 스레드를 사용하지 않고도 스트림을 병렬로 처리할 수 있다.
- 간단하게 스트림을 병렬로 처리할 수 있지만 항상 병렬 처리가 빠른 것은 아니다. 병렬 소프트웨어 동작 방법과 성능은 직관적이지 않을 때가 많으므로 병렬 처리를 사용했을 때 성능을 직접 측정해봐야 한다.
- 병렬 스트림으로 데이터 집합을 병렬 실행할 때 특히 처리해야 할 데이터가 아주 많거나 각 요소를 처리하는 데 오랜 시간이 걸릴 때 성능을 높일 수 있다.
- 가능하면 기본형 특화 스트림을 사용하는 등 올바른 자료구조 선택이 어떤 연산을 병렬로 처리하는 것보다 성능적으로 더 큰 영향을 미칠 수 있다.
- 포크/조인 프레임워크에서는 병렬화 할 수 있는 태스크를 작은 태스크로 분할한 다음에 분할된 태스크를 각각의 스레드로 실행하며 서브태스크 각각의 결과를 합쳐서 최종 결과를 생산한다.
- Splitter는 탐색하려는 데이터를 포함하는 스트림을 어떻게 병렬화 할 것인지 정의한다.