동시성과 병렬성 복습 : https://seamless.tistory.com/42

Java 7 이 등장하기 이전의 데이터 컬렉션 병렬 처리의 어려움

1. 데이터를 서브파트로 분할
2. 분할된 서브파트를 각각의 스레드로 할당
3. 스레드로 할당한 다음에는 의도치 않은 레이스 컨디션이 발생하지 않도록 적절한 동기화
4. 마지막으로 부분 결과를 합침

Java 7은 더 쉽게 병렬화를 수행하면서 에러를 최소화할 수 있도록 포크/조인 프레임워크를 제공.

병렬 스트림

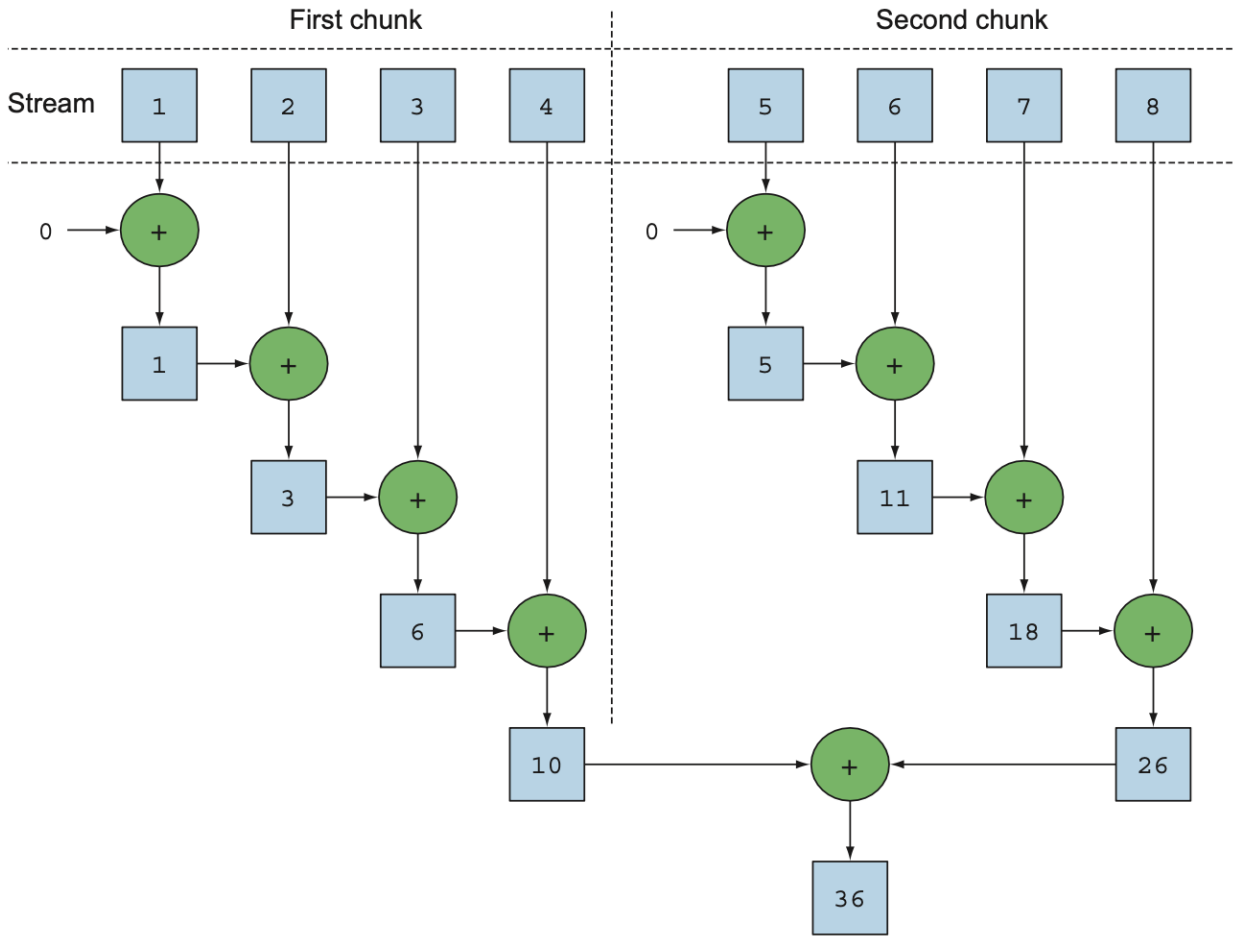
병렬 스트림이란 각각의 스레드에서 처리할 수 있도록 스트림 요소를 여러 청크로 분할한 스트림.

청크 : https://en.wikipedia.org/wiki/Chunk_(information)
fragment of information

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

```
1 usage
public static long parallelSum(long n) {
    return Stream.iterate( seed: 1L, i → i + 1)
        .limit(n)
        .parallel()
        .reduce( identity: 0L, Long::sum);
}
```

일반 반복 코드와 다른 점은
스트림이 여러 청크로 분할되어 있다는 점이다. 따라서 리듀싱 연산을 여러 청크에 병렬로 수행할 수 있다. 마지막으로 리듀싱 연산으로 생성된 부분 결과를 다시 리듀싱 연산으로 합쳐서 전체 스트림의 리듀싱 결과를 도출한다.



병렬 스트림은 내부적으로 ForkJoinPool을 사용한다.

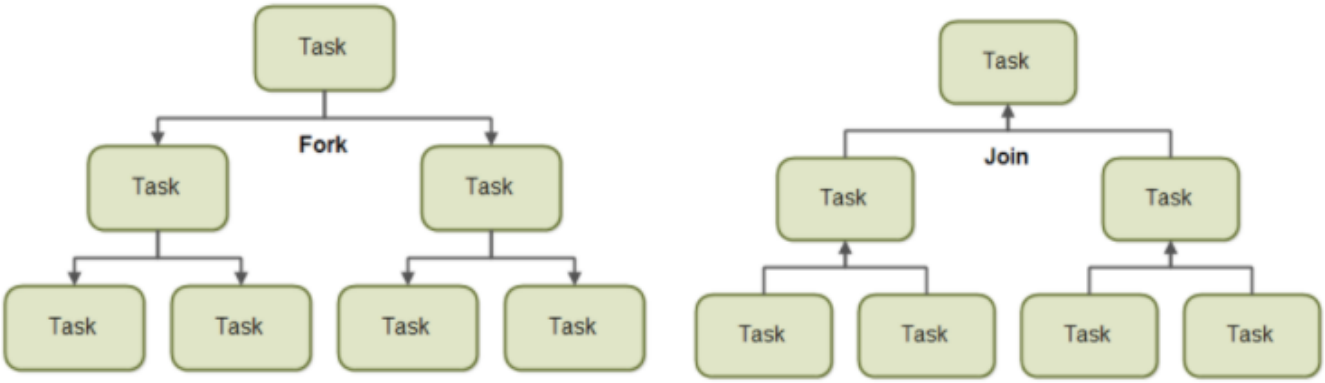
기본적으로 ForkJoinPool은 프로세서 수, `Runtime.getRuntime().availableProcessors()`가 반환하는 값에 상응하는 스레드를 갖는다.

`ForkJoinPool`

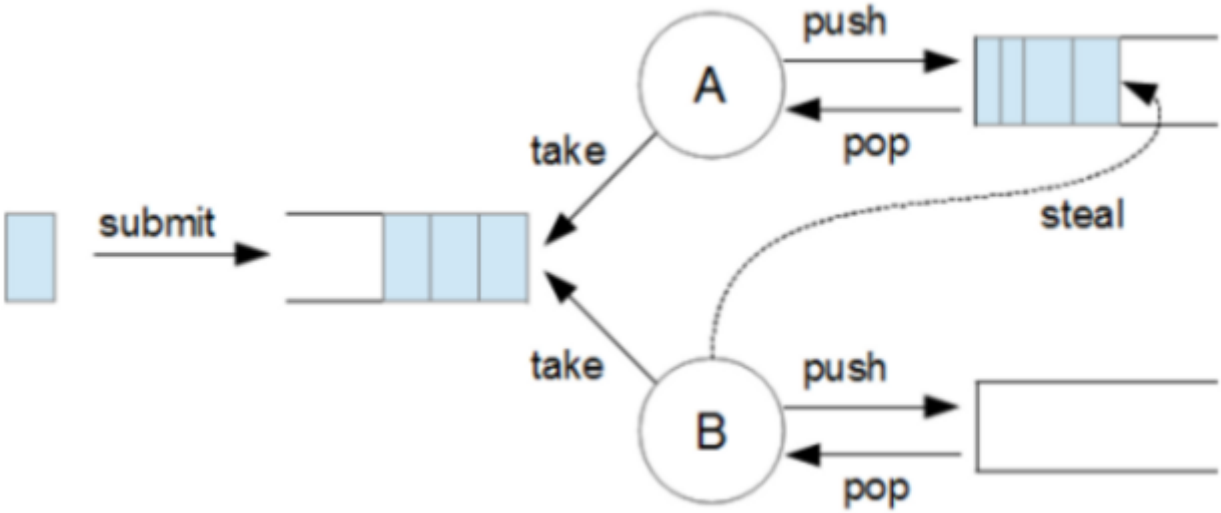
```
java.lang.Object
                java.util.concurrent.AbstractExecutorService
                                    java.util.concurrent.ForkJoinPool
```

ForkJoinPool이란 Java 7부터 사용가능한 Java Concurrency 툴이며 **동일한 작업을 여러 개의 Sub Task로 분리(Fork) 하여 각각 처리하고, 이를 최종적으로 합쳐서(Join) 결과를 만들어내는 방식이다.**

Fork : https://ko.wikipedia.org/wiki/%ED%8F%AC%ED%81%AC_(%EC%86%8C%ED%94%84%ED%8A%B8%EC%9B%A8%EC%96%B4_%EA%B0%9C%EB%B0%9C) 실행 중인 프로세스가 그 자신을 2개의 (거의)동일한 사본으로 나뉘게 하여 각기 다른 작업을 수행케 한다는 의미



Fork 를 통해 태스크를 분담하고, Join을 통해서 태스크를 취합한다.



하나의 작업 큐를 가지고 있으며, 이러한 작업들을 ForkJoinPool에서 관리하는 여러 스레드에서 접근하여 작업을 처리한다.
서로 작업을 하려고 큐에서 작업을 가져가며, 각 스레드들은 부모 큐에서 가져간 작업들을 내부 큐(inbound queue)에 담아 관리한다.
스레드들은 서로의 큐에 접근하여 작업들을 가져가 처리한다.
이러한 방법들은 유휴 스레드가 없도록 하기 위해 도입되었다. => **work stealing 기법**

위의 그림 처럼 스레드들이 관리하는 큐는 Deque(덱)이기 때문에 B스레드에서 A스레드의 작업을 뺏어간다.

스레드 자신의 task queue로 Deque(덱)을 사용한다.
각 스레드는 덱의 한쪽 끝에서만 일한다. 나머지 한쪽 끝에는 작업을 훔치러 온 다른 스레드가 접근한다.

부모 작업은 Task를 분할하여 fork함으로써 자식 스레드에서 처리되며 join을 통해 포함된다.

ForkJoin 처리를 위한 Task 클래스를 만드려면 추상 클래스인 `RecursiveTask<T>` 또는 `RecursiveAction` 을 상속 받아야 된다. `RecursiveTask<T>` 와 `RecursiveAction`의 차이는 처리 작업의 리턴 값 유무이다.

스트림 성능 측정

JVM으로 실행되는 프로그램을 벤치마크하는 작업은 쉽지 않다.

1. 핫스팟이 바이트코드를 최적화하는데 필요한 준비 시간
2. 가비지 컬렉터로 인한 오버헤드

등과 같은 여러 많은 요소를 고려해야 하기 때문이다.

```
Benchmark                                  Mode  Cnt   Score    Error  Units
ParallelStreamBenchmark.iteraviteSum       avgt    4   3.571 ±  0.016  ms/op
ParallelStreamBenchmark.parallelRangedSum  avgt    4   4.659 ± 12.834  ms/op
ParallelStreamBenchmark.parallelSum        avgt    4  73.377 ± 42.564  ms/op
ParallelStreamBenchmark.rangedSum          avgt    4   3.950 ±  0.273  ms/op
ParallelStreamBenchmark.sequentialSum      avgt    4  64.069 ±  9.072  ms/op
```

테스트 결과 순차 스트림보다 병렬 스트림이 느린 것을 확인할 수 있었다.

- 반복 결과로 박싱된 객체가 만들어지므로 숫자를 더하려면 언박싱을 해야 한다.Long->long
- 반복 작업은 병렬로 수행할 수 있는 `독립 단위로 나누기`가 어렵다.

iterate 연산은 이전 연산의 결과에 따라 다음 함수의 입력이 달라지기 때문에 청크로 분할하기가 어렵다.

리듀싱을 시작하는 시점에 전체 숫자 리스트가 준비되지 않았으므로 스트림을 병렬로 처리할 수 있도록 청크로 분할할 수 없다.
스트림이 병렬로 처리되도록 지시했고 각각의 합계가 다른 스레드에서 수행되었지만 결국 순차처리 방식과 크게 다른 점이 없으므로 **스레드를 할당하는 오버헤드만 증가하게 된다.**

-> 더 특화된 메서드 사용으로 해결(LongStream.rangeClosed)

- 오토박싱 / 언박싱과 관련한 오버헤드는 크다.
- 쉽게 청크로 분할할 수 있는 숫자 범위를 생산한다.

```java
@Benchmark
public long sequentialSum() {
    return Stream.iterate( seed: 1L, i -> i + 1)
        .limit(REPEAT_NUMBER)
        .reduce( identity: 0L, Long::sum);
}

@Benchmark
public long rangedSum() {
    return LongStream.rangeClosed(1, REPEAT_NUMBER)
        .reduce( identity: 0L, Long::sum);
}
```

벤치마크 점수 약 16.22배 차이가 나타남을 알 수 있다.

-> 상황에 따라서는 알고리즘을 병렬화 하는 것보다 적절한 자료구조를 선택하는 것이 더 중요하다는 사실을 단적으로 보여준다.

```java
@Benchmark
public long parallelRangedSum() {
    return LongStream.rangeClosed(1, REPEAT_NUMBER)
        .parallel()
        .reduce( identity: 0L, Long::sum);
}
```

병렬 스트림을 적용한 코드는 다음과 같다.

병렬화가 완전 공짜는 아니다.

병렬화를 이용하려면
스트림을 재귀적으로 분할해야 하고,
각 서브스트림을 서로 다른 스레드의 리듀싱 연산으로 할당하고,
이들 결과를 하나의 값으로 합쳐야 한다.

멀티코어 간의 데이터 이동은 생각보다 비싸므로, 따라서 코어 간에 데이터 전송 시간보다 훨씬 오래 걸리는 작업만 병렬로 다른 코어에서 수행하는 것이 바람직하다.

`병렬 스트림의 올바른 사용법`

병렬 스트림을 잘못 사용하면서 발생하는 많은 문제는 **공유된 상태를 바꾸는 알고리즘**을 사용하기 때문에 일어난다.

```
1 usage
public static long sideEffectSum(long n) {
    Accumulator accumulator = new Accumulator();
    LongStream.rangeClosed(1, n).forEach(accumulator::add);
    return accumulator.total;
}

4 usages
public static class Accumulator {
    3 usages
    private long total = 0;

    public void add(long value) {
        total += value;
    }
}
```

total += value는 아토믹 연산이 아니다.

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

병렬 스트림이 올바르게 동작하려면 공유된 가변 상태를 피해야 한다.

병렬 스트림 효과적으로 사용하기

양을 기준으로 병렬 스트림 사용을 결정하는 것은 적절하지 않다.

- 확신이 서지 않으면 직접 측정하라. 적절한 벤치마크로 직접 성능을 측정하는것이 바람직하다.
- 박싱을 주의하라. 자동 박싱과 언박싱은 성능을 크게 저하시킬 수 있는 요소이다. IntStream, LongStream, DoubleStream 와 같은 기본형 특화 스트림을 이용하는 것이 좋다.
- 순차 스트림보다 병렬 스트림에서 성능이 떨어지는 연산이 있다. **limit이나 findFirst**처럼 요소의 순서에 의존하는 연산을 병렬 스트림에서 수행하려면 비싼 비용을 치러야 한다. 예를 들어 findAny는 요소의 순서와 상관없이 연산하므로 findFirst보다 성능이 좋다. 정렬된 스트림에 unordered를 호출해서 비정렬된 스

트림을 얻을 수 있는데, 스트림에 N개 요소가 있을 때 요소의 순서가 상관없다면 비정렬된 스트림에 limit를 호출하는 것이 더 효율적이다.

- 스트림에서 수행하는 전체 파이프라인 연산 비용을 고려하라. 처리해야 할 요소 수(N) * 하나의 요소를 처리하는데 드는 비용(Q). Q가 높아질수록 병렬 스트림으로 성능 개선할 가능성이 높아진다.
- 소량의 데이터에서는 병렬 스트림이 도움이 되지 않는다.
- 스트림을 구성하는 자료구조가 적절한지 확인하라. 예를 들어 ArrayList를 LinkedList 보다 효율적으로 분할할 수 있다. LinkedList를 분할하려면 모든 요소를 탐색해야 함. ArrayList는 요소를 탐색하지 않고도 리스트를 분할할 수 있다. 또한 range 팩토리 메서드로 만든 기본형 스트림도 쉽게 분해할 수 있다. 마지막으로 커스텀 Spliterator를 구현해서 분해 과정을 완벽하게 제어할 수 있다.
- 스트림의 특성과 파이프라인의 중간 연산이 스트림의 특성을 어떻게 바꾸는지에 따라 분해 과정의 성능이 달라질 수 있다. 예를 들어 SIZED 스트림(크기가 알려진 소스) 는 정확히 같은 크기의 두 스트림으로 분할할 수 있으므로 효과적으로 스트림을 병렬 처리할 수 있다. 반면 필터 연산이 있으면 스트림의 길이를 예측할 수 없으므로 효과적으로 스트림을 병렬 처리할 수 있을지 알 수 없게 된다.
- 최종 연산의 병합 과정(예를 들어 Collector의 combiner 메서드) 비용을 살펴보라. 병합 과정의 비용이 비싸다면 병렬 스트림으로 얻은 성능의 이익이 서브스트림의 부분 결과를 합치는 과정에서 상쇄될 수 있다.

스트림 소스와 분해성

ArrayList, Intstream.range - 훌륭함

HashSet, Treeset = 좋음

LinkedList, Stream.iterate - 나쁨

이펙티브 자바 item 48 . 스트림 병렬화는 주의해서 적용하라

동시성 프로그래밍을 할 때는 안전성(safety)와 응답 가능(liveness) 상태를 유지하기 위해 애써야 한다.

스트림 라이브러리가 파이프라인을 병렬화하는 방법을 찾아내지 못하면 응답 불가 상태에 빠질 수 있다.
(liveness failure)

```
public static void main(String[] args) {
        primes().map(p -> TWO.pow(p.intValueExact()).subtract(ONE))
        .filter(mersenne -> mersenne.isProbabblePrime(50))
        .limit(20)
        .forEach(System.out::println);
}
static Stream<BigInteger> primes() {
```

```
        return Stream.iterate(TWO, BigInteger::nextProbablePrime);
}
```

환경이 아무리 좋더라도 **데이터 소스가 Stream.iterate거나 중간 연산으로 limit을 쓰면 파이프라인 병렬화로는 성능 개선을 기대할 수 없다.**

파이프라인 병렬화는 limit을 다룰 때 CPU 코어가 남는다면 원소를 몇 개 더 처리한 후 제한된 개수 이후의 결과를 버려도 아무런 해가 없다고 가정한다.

대체로 **스트림의 소스가 ArrayList, HashMap, HashSet, ConcurrentHashMap의 인스턴스거나 배열, int 범위, long 범위일 때 병렬화의 효과가 가장 좋다.**
-> 이 자료구조들은 모두 데이터를 원하는 크기로 정확하고 손쉽게 나눌 수 있어서 일을 다수의 스레드에 분배하기에 좋다.

이 자료구조들의 또 다른 중요한 공통점들은 원소들을 순차적으로 실행할 때의 참조 지역성(locality of reference)이 뛰어나다는 것이다.
Cache hit rate :
https://ndlessrain.tistory.com/entry/%EC%BA%90%EC%8B%9C%EB%A9%94%EB%AA%A8%EB%A6%AC-%EC%A0%81%EC%A4%91%EB%A5%A0Hit-Rate

스트림 파이프라인의 종단 연산의 동작 방식 역시 병렬 수행 효율에 영향을 준다.
종단 연산에서 수행하는 작업량이 파이프라인 전체 작업에서 상당 비중을 차지하면서 순차적인 연산이라면 파이프라인 병렬 수행의 효과는 제한될 수밖에 없다.

종단 연산 중 병렬화에 가장 적합한 것은 축소(reduction)다.
축소는 파이프라인에서 만들어진 모든 원소를 하나로 합치는 작업으로, **Stream의 reduce 메서드 중 하나, 혹은 min, max, count, sum** 같이 완성된 형태로 제공되는 메서드 중 하나를 선택해 수행한다.

anyMatch, allMatch, noneMatch 처럼 조건에 맞으면 바로 반환되는 메서드도 병렬화에 적합하다.

반면, 가변 축소(mutable reduction)을 수행하는 Stream의 collect 메서드는 병렬화에 적합하지 않다. 컬렉션들을 합치는 부담이 크기 때문이다.

스트림을 잘못 병렬화하면 (응답 불가를 포함해) 성능이 나빠질 뿐만 아니라 결과 자체가 잘못되거나 예상 못한 동작이 발생할 수 있다.

결과가 잘못되거나 오작동하는 것은 안전 실패(safety failure)라 한다.

출력 순서를 순차 버전처럼 정렬하고 싶다면 종단 연산 forEach 를 forEachOrdered로 바꿔주면 된다.

스트림 안의 원소 수 * 원소당 수행되는 코드 줄 수 = 최소 수십만 이어야 성능 향상을 맛볼 수 있다.

보통은 병렬 스트림 파이프라인도 공통의 ForkJoinPool에서 수행되므로(즉, 같은 스레드 풀을 사용하므로) 잘못된 파이프라인 하나가 시스템의 다른 부분에까지 악영향을 줄 수 있다.

무작위 수들로 이뤄진 스트림을 병렬화하려면 **ThreadLocalRandom(혹은 Random) 보다 SplittableRandom** 인스턴스를 활용하자. 마지막으로 그냥 Random은 모든 연산을 동기화하기 때문에 병렬 처리하면 최악의 성능을 보일 것이다.

`7.2 포크/조인 프레임워크`

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

포크/조인 프레임워크에서는 서브태스크를 스레드 풀(ForkJoinPool)의 작업자 스레드에 분산 할당하는 ExecutorService 인터페이스를 구현합니다.

스레드 풀 개념

- 스레드 풀은 작업 처리에 사용되는 스레드를 제한된 개수만큼 정해 놓고 작업 큐에 들어오는 작업들을 하나씩 스레드가 맡아 처리합니다.
- 작업 처리가 끝난 스레드는 다시 작업 큐에서 새로운 작업을 가져와 처리합니다.
- 작업 처리 요청이 폭증해도 작업 큐라는 곳에 작업이 대기하다가 여유가 있는 스레드가 그것을 처리하므로 스레드의 전체 개수는 일정하며 애플리케이션의 성능도 저하되지 않습니다.

7.2.1 RecursiveTask 활용

스레드 풀을 이용하려면 `RecursiveTask<R>` 의 서브클래스를 만들어야 합니다.

여기서 R은 병렬화된 태스크가 생성하는 결과 형식입니다. 결과가 없을 때는 `RecursiveAction` 형식입니다. (결과가 없더라도 다른 비지역 구조를 바꿀 수 있습니다)

`RecursiveTask` 를 정의하려면 추상 메서드 compute 를 구현해야 합니다.

```
protected abstract R compute();
```

compute 메서드는 태스크를 서브태스크로 분할하는 로직과

더 이상 분할할 수 없을 때 개별 서브태스크의 결과를 생산할 알고리즘을 정의합니다.

따라서 대부분의 compute 메서드 구현은 다음과 같은 의사코드 형식을 유지합니다.

```
if (태스크가 충분히 작거나 더 이상 분할할 수 없으면) {
순차적으로 태스크 계산
} else {
태스크를 두 서브태스크로 분할
태스크가 다시 서브태스크로 분할되도록 이 메서드를 재귀적으로 호출함
모든 서브태스크의 연산이 완료될 때까지 기다림
각 서브태스크의 결과를 합침
}
```

이 알고리즘은 분할 후 정복 알고리즘의 병렬화 버전입니다.

분할 정복 알고리즘은 그대로 해결할 수 없는 문제를 작은 문제로 분할하여 문제를 해결하는 방법이나 알고리즘입니다.

`ForkJoinSumCalculator`

```
public class ForkJoinSumCalculator extends RecursiveTask<Long> {
private static final long THRESHOLD = 10_000; // 이 값 이하의 서브태스크는 더 이상 분할할 수 없다.
private final long[] numbers;
private final int start; // 이 서브태크스에서 처리할 배열의 초기 위치
private final int end; // 이 서브태스크에서 처리할 배열의 최종 위치

public ForkJoinSumCalculator(long[] numbers) { // 메인 태스크를 생성할 때 사용할 공개 생성자
this(numbers, 0, numbers.length);
}

private ForkJoinSumCalculator(long[] numbers, int start, int end) { // 메인 태스크의 서브태스크를 재귀적으로 만들 때 사용할 비공개 생성자
this.numbers = numbers;
this.start = start;
this.end = end;
}
```

```
@Override
protected Long compute() {
int length = end - start;
if (length <= THRESHOLD) { // 기준값과 같거나 작으면 순차적으로 결과를 계산한다.
return computeSequentially();
}

ForkJoinSumCalculator leftTask = new ForkJoinSumCalculator(numbers, start,
start + length / 2);
leftTask.fork();

ForkJoinSumCalculator rightTask = new ForkJoinSumCalculator(numbers, start +
length / 2, end);

Long rightResult = rightTask.compute();
Long leftResult = leftTask.join();
return leftResult + rightResult;
}


private long computeSequentially() {
        long sum = 0;
        for (int i = start; i < end; ++i) {
        sum += numbers[i];
}
return sum;
}

public long computeSequentiallyWithStream() {
        return Arrays.stream(numbers)
                                .sum();
}
}
```

위 메서드는 n까지의 자연수 덧셈 작업을 병렬로 수행하는 방법을 더 직관적으로 보여줍니다.

다음 코드처럼 ForkJoinSumcalculator의 생성자로 원하는 수의 배열을 넘겨줄 수 있습니다.

```
public static long forkJoinSum(long n) {

// n까지의 자연수 덧셈 작업을 병렬로 수행하는 방법

long[] numbers = LongStream.rangeClosed(1, n).toArray();
```

```
ForkJoinTask<Long> task = new ForkJoinSumCalculator(numbers);

return new ForkJoinPool().invoke(task);

}
```

LongStream으로 n까지의 자연수를 포함하는 배열을 생성했습니다.

그리고 생성된 배열을 ForkJoinSumCalculator의 생성자로 전달해서 ForkJoinTask를 만들었습니다.

마지막으로 생성한 태스크를 새로운 ForkJoinPool의 invoke 메서드로 전달했습니다.

ForkJoinPool에서 실행되는 마지막 invoke 메서드의 반환값은 ForkJoinSumCalculator에서 정의한 태스크의 결과가 됩니다.

일반적으로 애플리케이션에서는 둘 이상의 ForkJoinPool을 사용하지 않습니다.

즉, 소프트웨어의 필요한 곳에서 언제든 가져다 쓸 수 있도록 ForkJoinPool을 한 번만 인스턴스화해서 **정적 필드에 싱글턴으로 저장합니다.**

ForkJoinPool을 만들면서 인수가 없는 디폴트 생성자를 이용했는데, 이는 JVM에서 이용할 수 있는 모든 프로세서가 자유롭게 풀에 접근할 수 있음을 의미합니다.

더 정확하게는 Runtime.availableProcessors의 반환값으로 풀에 사용할 스레드 수를 결정합니다.

`사용할 수 있는 프로세서`라는 이름과는 달리 실제 프로세서 외에 `하이퍼스레딩과 관련된 가상 프로세서도 개수에 포함합니다.

ForkJoinSumCalculator 실행

ForkJoinSumCalculator를 ForkJoinPool로 전달하면 풀의 스레드가 ForkJoinSumCalculator의 compute 메서드를 실행하면서 작업을 수행합니다.

compute 메서드는 병렬로 실행할 수 있을만큼 태스크의 크기가 충분히 작아졌는지 확인하며, 아직 태스크의 크기가 크다고 판단되면

숫자 배열을 반으로 분할해서 두 개의 새로운 ForkJoinSumCalculator로 할당한다.

그러면 다시 ForkJoinPool이 새로 생성된 ForkJoinSumCalculator를 실행한다.

이 과정이 재귀적으로 반복되면서 주어진 조건(예제에서는 덧셈을 수행할 항목이 만 개 이하)을 만족할 때까지 태스크 분할을 반복한다.

이제 각 서브태스크는 순차적으로 처리되며 포킹 프로세스로 만들어진 이진트리의 태스크를 루트에서 역순으로 방문합니다.

즉, 각 서브태스크의 부분 결과를 합쳐서 태스크의 최종 결과를 계산합니다.

7.2.2 포크/조인 프레임워크를 제대로 사용하는 방법

- join 메서드를 태스크에 호출하면 태스크가 생산하는 결과가 준비될 때까지 호출자를 블록시킨다.따라서 두 서브태스크가 모두 시작된 다음에 join을 호출해야 한다.
- RecursiveTask 내에서는 ForkJoinPool의 invoke 메서드를 사용하지 말아야 한다. 대신 compute나 fork 메서드를 직접 호출할 수 있다.
- 서브태스크에 fork 메서드를 호출해서 ForkJoinPool의 일정을 조절할 수 있다. 왼쪽 작업과 오른쪽 작업 모두에 fork를 호출하는 것보다는

한쪽 작업에는 fork 보다는 compute를 호출하는 것이 효율적이다. 두 서브태스크의 한 스레드에는 같은 스레드를 재사용할 수 있으므로 오버헤드를 피할 수 있다.

- 포크/조인 프레임워크를 이용하는 병렬 계산은 디버깅하기 어렵다.
- 멀티코어에 포크/조인 프레임워크를 사용하는 것이 순차 처리보다 무조건 빠를 생각은 버려야 한다.

병렬 처리로 성능을 개선하려면 태스크를 여러 독립적인 서브태스크로 분할할 수 있어야 한다.

각 서브태스크의 실행시간은 새로운 태스크를 포킹하는 데 드는 시간보다 길어야 한다.

7.2.3 작업 훔치기

work stealing

작업 훔치기 기법에서는 ForkJoinPool의 모든 스레드를 거의 공정하게 분할한다.

각각의 스레드는 자신에게 할당된 태스크를 포함하는 이중 연결 리스트를 참조하면서 작업이 끝날 때마다 큐의 헤드에서 다른 태스크를 가져와서 작업을 처리한다.

다른 스레드는 바쁘게 일하고 있는데 한 스레드는 할일이 다 떨어진 상황

-> 할일이 없어진 스레드가 유휴 상태(어떠한 프로그램에 의해서도 사용되지 않는 상태)로 바뀌는 것이 아니라 다른 스레드 큐의 꼬리에서 작업을 훔쳐온다.

모든 태스크가 작업을 끝낼 때까지, 즉 모든 큐가 빌 때까지 이 과정을 반복한다. 따라서 태스크의 크기를 작게 나누어야 작업자 스레드 간의 작업부하를 비슷한 수준으로 유지할 수 있다.

다음 절에서는 스트림을 자동으로 분할하는 기법인 Spliterator 를 설명합니다.