

7. 포크조인 프레임워크 - 숏가락 조인은 없나

Java 병렬 데이터 처리와 성능 최적화

외부 반복을 내부 반복으로 바꾸면 네이티브 자바 라이브러리가 스트림 요소의 처리를 제어할 수 있다.

컴퓨터의 멀티코어를 활용해서 파이프라인 연산을 실행할 수 있다는 점이 가장 중요한 특징이다.

목차


1. [병렬 스트림 기본 개념](#)
2. [Fork/Join 프레임워크](#)
3. [Spliterator 인터페이스](#)

Java 7 이전의 병렬 처리 문제점

Java 7이 등장하기 전에는 데이터 컬렉션을 병렬로 처리하기가 어려웠다.

// 기존 방식의 병렬 처리 과정

1. 데이터를 서브 파트로 분할
2. 분할된 서브 파트를 각각의 스레드로 할당
3. 의도치 않은 레이스 컨디션이 발생하지 않도록 적절한 동기화 추가
4. 부분 결과를 합치기

 **해결책:** Java 7은 더 쉽게 병렬화를 수행하면서 에러를 최소화할 수 있도록 **포크/조인 프레임워크** 기능을 제공한다.

1. 병렬 스트림

병렬 스트림이란?

컬렉션에 `parallelStream` 을 호출하면 **병렬 스트림**이 생성된다. 병렬 스트림이란 각각의 스레드에서 처리할 수 있도록 스트림 요소를 여러 청크로 분할한 스트림이다.

따라서 병렬 스트림을 이용하면 모든 멀티코어 프로세서가 각각의 청크를 처리할 수 있도록 할당할 수 있다.

1.1 순차 스트림을 병렬 스트림으로 변환하기

순차 스트림에 `parallel()` 메소드를 호출하면 기존의 함수형 리듀싱 연산이 병렬로 처리된다.

```
public long parallelSum(long n) {
    return Stream.iterate(1L, i -> i + 1)
        .limit(n)
        .parallel() // 병렬 스트림으로 변환
        .reduce(0L, Long::sum);
}
```

💡 동작 원리

- 리듀싱 연산으로 스트림의 모든 숫자를 더한다
- 순차방식과 다른 점은 여러 청크로 분할되어 있다는 것이다
- 여러 청크에 병렬로 수행하여 마지막에 리듀싱 연산으로 생성된 부분 결과를 다시 리듀싱 연산으로 합쳐서 전체 스트림의 리듀싱 결과를 도출한다

🔄 **순차 스트림으로 전환** 반대로 `sequential()` 로 병렬 스트림을 순차 스트림으로 바꿀 수 있다.

```
stream.parallel()
    .filter(...)
    .sequential()
    .map(...)
    .parallel()
    .reduce();
```

⚠️ **주의:** 최종적으로 호출된 메소드가 전체 파이프라인에 영향을 미친다. 위 예제에서 파이프라인의 마지막 호출은 `parallel()` 이므로 파이프라인은 전체적으로 병렬로 실행된다.

⚠️ 병렬화 사용 시 주의사항

병렬화가 완전 공짜는 아니라는 사실을 기억하자.

병렬화를 이용하려면:

- 스트림을 재귀적으로 분할해야 한다
- 각 서브 스트림을 서로 다른 스레드의 리듀싱 연산으로 할당한다
- 이들 결과를 하나의 값으로 합쳐야 한다

멀티코어 간의 데이터 이동은 우리 생각보다 비싸다. 따라서 코어 간에 데이터 전송시간보다 훨씬 오래 걸리는 작업만 병렬로 다른 코어에서 수행하는 것이 바람직하다.

1.2 병렬 스트림의 올바른 사용법

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


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

```

}

public class Accumulator {
    public long total = 0;
    public void add(long value) {
        total += value;
    }
}

```

 **위험:** 위 코드는 본질적으로 순차 실행할 수 있도록 구현되어 있으므로 병렬로 실행하면 참사가 일어난다. 특히 `total` 을 접근할 때마다 다수의 스레드에서 동시에 데이터에 접근하는 **데이터 레이스 문제**가 일어난다.

주요 문제점들

문제1. 공유된 상태를 바꾸는 알고리즘 사용

Error parsing Mermaid diagram!

Cannot read properties of null (reading 'getBoundingClientRect')

핵심 개념

- **Data Race:** 동기화 없이 같은 변수에 동시 접근
- **Race Condition:** 실행 타이밍에 따라 결과가 바뀌는 더 넓은 개념

해결방안

- 가급적 외부 상태에 손대지 말아야 한다
- 람다는 순수함수처럼 입력만 보고 결과만 반환해야 한다
- 로깅, 카운팅 같은 부작용도 지양해야 한다

1.3 병렬 스트림을 효과적으로 사용하기

순차에서 병렬로 바꾸는 것은 쉽지만 무조건 병렬로 바꾸는 것은 권장하지 않는다. 항상 빠른 것도 아니고 수행과정 또한 투명하지 않은 경우가 많다.

문제 2. 박싱 주의

```

// ❌ 성능 저하
Stream<Integer> numbers = Stream.iterate(1, i -> i + 1);

// ✅ 권장
IntStream numbers = IntStream.rangeClosed(1, n);

```

자동 박싱/언박싱은 성능 저하 요소 중 하나이다. 기본형 특화 스트림(`IntStream`, `LongStream`, `DoubleStream`)을 사용하는 것이 좋다.

문제 3. 순서 의존성

순차 스트림보다 병렬 스트림에서 성능이 떨어지는 연산이 있다. 특히 `limit`, `findFirst` 처럼 순서에 의존하는 연산을 병렬 스트림에서 수행하려면 비싼 비용을 치러야 한다.

```
// 성능 비교
findAny()      // ✅ 순서 상관없음 - 병렬에서 유리
findFirst()    // ⚠️ 순서 의존적 - 병렬에서 불리

// 최적화 팁
unorderedStream.limit(n) // ✅ 비정렬 스트림에서 limit 사용
```

문제 4. 전체 파이프라인 연산 비용 고려

N: 처리해야 하는 요소 수
Q: 하나의 요소를 처리하는 데 드는 비용

전체 스트림 파이프라인 처리 비용 = $N \times Q$

💡 결론: Q가 높아지면 병렬로 개선 가능하다.

문제 5. 자료구조의 분할 효율성

자료구조	분할 효율성	이유
ArrayList	✅ 높음	인덱스 기반 접근으로 효율적 분할
LinkedList	❌ 낮음	모든 요소를 탐색해야 함
HashSet	⚠️ 보통	해시 기반이지만 순서 없음
TreeSet	✅ 높음	균형 트리 구조

2. Fork/Join 프레임워크

🔧 기본 개념

"병렬화" 할 수 있는 작업을 재귀적으로 작은 작업으로 분할한 다음에 서브 태스크 각각의 결과를 합쳐 전체 결과를 만들도록 설계된다.

Error parsing Mermaid diagram!

Cannot read properties of null (reading 'getBoundingClientRect')

🏗️ 주요 구성 요소

ExecutorService 계층

```

ExecutorService
├── ThreadPoolExecutor    // 일반적인 스레드 풀
├── ForkJoinPool          // Fork/Join 전용 스레드 풀
│   ├── RecursiveTask<V> // 결과를 반환하는 서브 태스크
│   └── RecursiveAction   // 결과가 없는 서브 태스크

```

ExecutorService란? 태스크들을 큐에 넣으면 스레드 풀들이 알아서 꺼내 실행해주는 작업 실행기이다.

주요 특징

- 매번 `new Thread()` 만들지 말고 `ThreadPool` 재사용해서 성능/안정성 확보
- 작업 제출/결과 받기, 취소/시간 제한, 스케줄링 같은 공통 기능 제공
- 서버에서 요청 처리/배치/IO 작업을 안전하게 병렬화

2.1 RecursiveTask 사용법

분할정복(쪼개기 → 각자 계산 → 합치기) 패턴을 사용하면서 결과가 필요한 포크 조인 작업에 쓰는 추상화 클래스이다.

- 결과가 필요하다면 `RecursiveTask<V>` 사용
- 결과가 필요 없다면 `RecursiveAction` 사용

기본 패턴

```

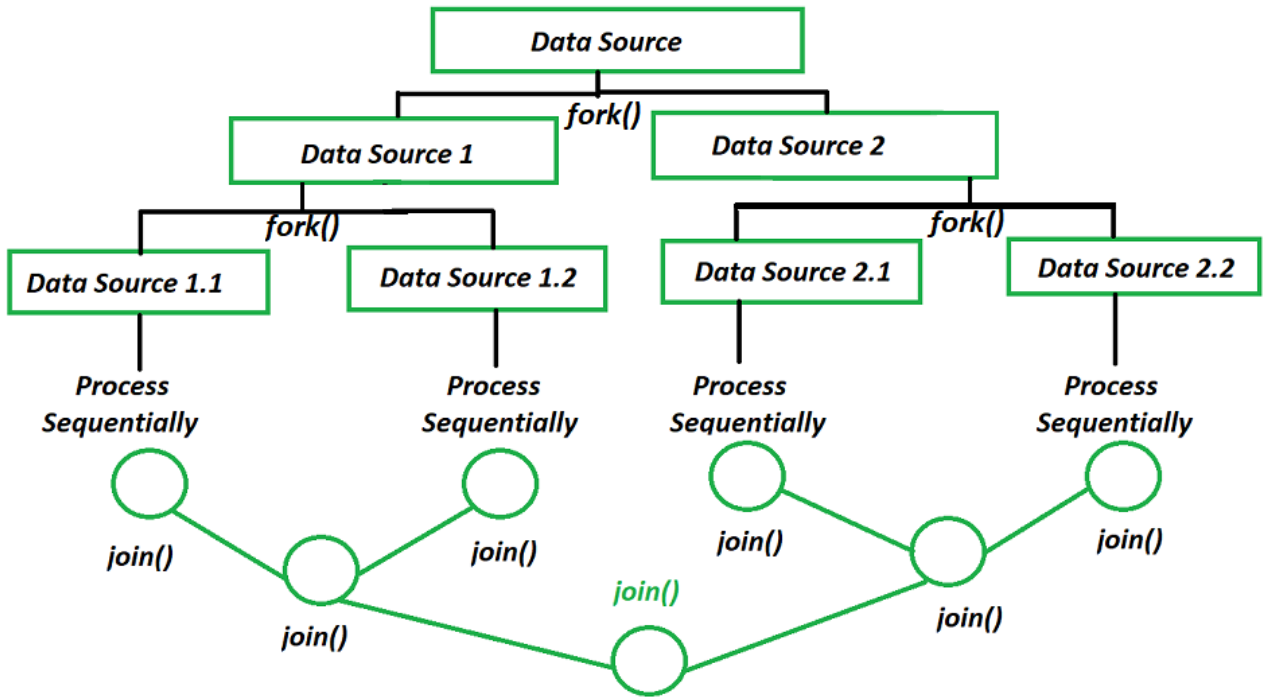
protected V compute() {
    if (작업이 충분히 작거나 더 이상 분할 불가) {
        순차적으로 작업 계산
    } else {
        작업을 두 개의 subtask로 분할
        왼쪽 = 새로운 RecursiveTask 생성
        오른쪽 = 새로운 RecursiveTask 생성

        왼쪽.fork();           // 비동기 실행
        오른쪽결과 = 오른쪽.compute(); // 현재 스레드에서 실행
        왼쪽결과 = 왼쪽.join(); // 결과 대기

        return 왼쪽결과 + 오른쪽결과;
    }
}

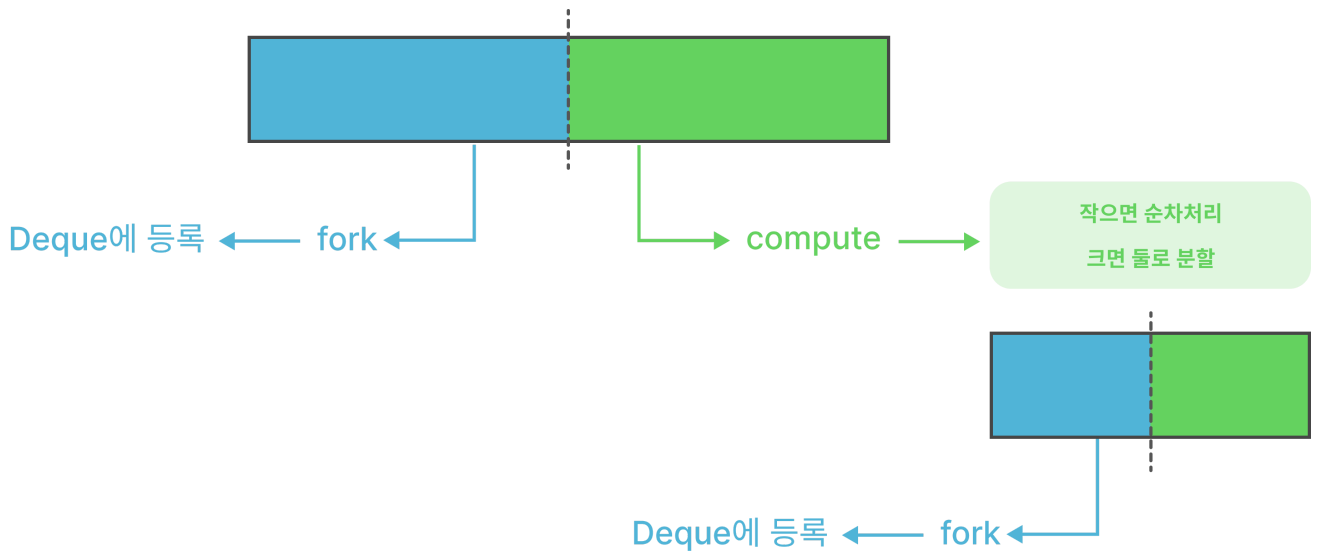
```

실행 순서: `fork()` → `compute()` → `join()`



전체 실행 흐름

1. compute()에서 크기 확인
2. 작으면 순차 처리, 크면 둘로 분할
3. 재귀적 분할 및 병합



🚀 작업 훔치기 (Work Stealing) 알고리즘

Deque와 작업 훔치기 메커니즘

Error parsing Mermaid diagram!

Cannot read properties of null (reading 'getBoundingClientRect')

핵심 원리

- 각 워커 스레드마다 전용 Deque가 있다 (하나의 공용 큐가 아님)
- 현재 워커는 자기 Deque의 head를 LIFO로 pop/push 한다
- 일 없는 다른 워커는 남의 Deque에서 작업을 훔친다(steal)
- 왼쪽을 fork로 큐에 밀고 오른쪽을 즉시 계산하면 훔쳐갈 작업을 남기고 내 스레드는 연속적으로 깊이 파고들며 일할 수 있다

부모-자식 결과 전달 방식

- 부모가 join() 으로 기다렸다가 자식의 반환 값을 받는 구조
- 부모 스레드가 join() 중이면 놀지 않고 다른 task를 도우며 기다릴 수 있다

⚙️ ForkJoinPool 설정

싱글톤 패턴 권장

일반적으로 애플리케이션에서는 둘 이상의 ForkJoinPool을 사용하지 않는다. 즉, 소프트웨어의 필요한 곳에서 언제든지 가져다 쓸 수 있게 ForkJoinPool을 한 번만 인스턴스화해서 정적 필드에 싱글톤으로 저장한다.

```
// 디폴트 생성자 사용 (CPU 코어 수만큼 워커 스레드 생성)
ForkJoinPool pool = new ForkJoinPool();
```

왜 싱글톤으로 할까?

1) CPU 코어 수 한정

- ForkJoinPool은 CPU 코어 수만큼 워커 스레드를 미리 띄운다
- JVM 안에 여러 개 만들면 스레드 수가 코어 수보다 n배 많아진다
- 따라서 context-switch, 캐시 미스, 메모리 사용 모두 증가한다

2) 작업 훔치기 극대화

- 풀을 공유하면 서로 다른 작업이라도 한쪽이 놀고 있으면 다른 작업을 훔쳐와 수행 가능
- 풀을 나누면 그 안에서만 작업 훔치기가 일어나 부하 분산이 제한적

2.2 포크/조인 프레임워크를 제대로 사용하는 방법

포크/조인 프레임워크는 쉽게 사용할 수 있는 편이지만 항상 주의를 기울여야 한다.

🚨 주의사항

1. join 메소드 호출 타이밍

```
// ❌ 잘못된 방법
left.fork();
leftResult = left.join(); // 즉시 join하면 순차적 실행
right.fork();
rightResult = right.join();

// ✅ 올바른 방법
```

```
left.fork();
right.fork();
leftResult = left.join();    // 두 작업 모두 시작된 후 join
rightResult = right.join();
```

2. invoke 메소드 사용 금지

- RecursiveTask 내에서는 ForkJoinPool의 invoke 메소드를 사용하지 않아야 한다
- 대신 compute, fork 메소드 호출이 가능하다
- 순차 코드에서 병렬 계산을 시작할 때만 invoke 사용한다

3. 효율적인 fork 패턴

```
// ✅ 권장 패턴
left.fork();                // 한쪽만 fork
rightResult = right.compute(); // 다른 쪽은 현재 스레드에서 실행
leftResult = left.join();
```

이유: 두 서브 태스크 중 한 태스크에는 같은 스레드를 재사용할 수 있으므로 풀에서 불필요한 태스크를 할당하는 오버헤드를 피할 수 있다.

4. 디버깅 어려움

포크/조인 프레임워크를 이용하는 병렬 계산은 디버깅하기 어렵다. fork 로 호출되는 다른 스레드에서 compute 를 호출하므로 스택 트레이스가 도움이 되지 않는다.

5. 성능 보장 없음

멀티코어에 포크/조인 프레임워크를 사용하는 것이 순차 처리보다 무조건 빠를 거라는 생각은 버려야 한다.

성능 개선 조건

- 태스크를 여러 독립적인 서브 태스크로 분할 가능해야 한다
- 각 서브태스크의 실행시간은 새로운 태스크 할당과 포킹하는데 드는 시간보다 길어야 한다

2.3 작업 흠치기 상세 분석

실제로는 코어 개수와 상관없이 적절한 크기로 분할된 많은 태스크를 포킹하는 것이 바람직하다.

현실적인 문제들

```
Error parsing Mermaid diagram!

Cannot read properties of null (reading 'getBBox')
```

문제 발생 원인

- 분할 기법이 효율적이지 않음
- 예기치 않은 디스크 접근 속도 저하

- 외부 서비스와 협력하는 과정에서 지연 발생

작업 훔치기 동작 흐름

1. fork() 호출

`task.fork()` ⇒ 현재 워커의 Deque 맨 앞(head)에 서브 태스크 push

2. 자기 할 일 소진

현재 워커의 Deque가 빌 때까지 head 쪽(LIFO)에서 pop하며 작업 수행

3. 남는 스레드 발생: 작업 훔치기 시도

- Idle 워커는 무작위로 다른 워커 하나 선택
- 그 워커의 Deque 뒤편(tail)을 단 하나만 atomically pop
- 오래된 작업 ⇒ 큰 덩어리일 가능성 높아 fork로 또다시 병렬성 생김

4. 훔친 작업 수행

새로 얻은 task도 같은 규칙으로 분할 및 실행

5. join()

모든 서브 태스크가 완료되면 부모 작업이 결과 집계

왜 효율적인가?

장점	설명
부하 균형	작업 분할 깊이/크기가 예측 불가해도 idle 스레드가 즉시 일감을 확보
낮은 락 경쟁	Deque는 앞뒤 접근 위치가 달라 락 없이 CAS 원자 연산만 사용
캐시 친화적	소유한 스레드는 최근에 분할한 작은 태스크를 LIFO로 바로 소비해 데이터 캐시 일관성 유지
스케일 아웃	코어 수가 늘어나도 중앙 큐를 공유하지 않아 병목이 완만

작업 훔치기 요약

원리

- 각 워커 스레드가 자기 전용 Deque에 작업 push
- 스스로 처리하다가 비면 `trySteal()` 로 다른 큐의 뒤편에서 작업 pop

언제 사용되는가?

- `parallelStream()` 내부
- `ForkJoinFramework` / `RecursiveTask` / `RecursiveAction`

장점

- 큐 고르게 사용 → 처리 시간 단축
- 큐 분산으로 경합 최소화

주의사항

- 블로킹 I/O 넣지 말 것 → 스레드 멈춰 큐 전체 정지
- 분할을 너무 작게 하지 말 것
- 공유 데이터는 스레드 안전하게

핵심 동작

- `fork` : 큰 작업을 둘로 잘라 자기가 만든 큐 head에 push
- `compute` : 자기 큐 앞에서 LIFO 순서로 처리
- `steal` : 다른 스레드가 놀면 꼬리에서 하나 pop
- `join` : 모든 서브 테스크 끝나면 결과 합산

3. Spliterator 인터페이스

기본 개념

Java 8에서는 **Spliterator**라는 새로운 인터페이스를 제공한다. Spliterator는 분할할 수 있는 반복자(**Split + Iterator**)이다.

Iterator처럼 Spliterator는 소스의 요소 탐색 기능을 제공하는 점은 동일하지만 **병렬 작업에 특화**되어 있다.

인터페이스 구조

```
public interface Spliterator<T> {  
  
    // 다음 요소가 있으면 소비하고 true 반환  
    boolean tryAdvance(Consumer<? super T> action);  
  
    // 가능하면 절반으로 분할해 새로운 Spliterator 반환  
    Spliterator<T> trySplit();  
  
    // 남은 요소 개수, 알 수 없으면 Long.MAX_VALUE  
    long estimateSize();  
  
    // 특성 정보 반환 (뒤에서 설명)  
    int characteristics();  
}
```

분할 과정

스트림을 여러 스트림으로 분할하는 과정은 재귀적으로 일어난다.

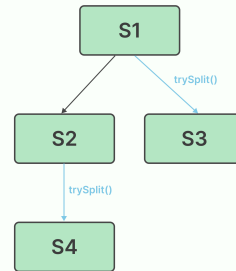
1. 첫 번째 Spliterator에 `trySplit` 호출하면 두 번째 Spliterator가 생성

2. 두 개의 Splitter에 trySplit 다시 호출하면 네 개의 Splitter가 생성
3. trySplit 결과가 null이 될 때까지 반복 → 더 이상 분할 불가

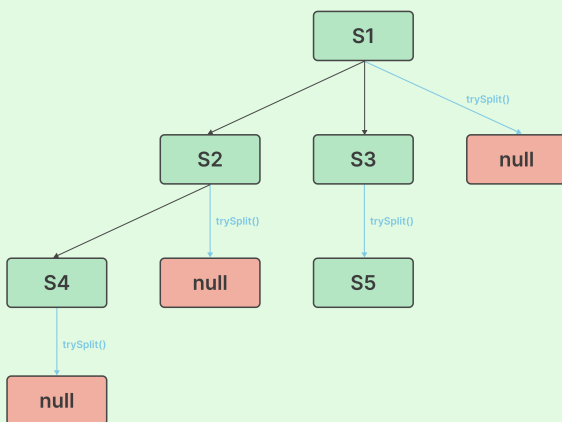
1.



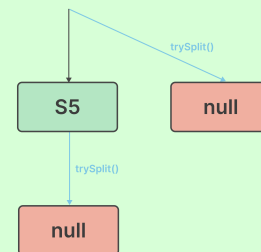
2.



3.



4.



🔖 특성(Characteristics)

주요 특성들

특성	설명	예시
ORDERED	요소에 정해진 순서가 있음	List, Stream.iterate
DISTINCT	모든 요소가 고유함 (x.equals(y) 항상 false)	Set
SIZED	크기가 알려진 소스 (estimateSize() 정확한 값 반환)	ArrayList.splititerator()
NONNULL	탐색한 모든 요소는 null이 아님	-
IMMUTABLE	요소 탐색 동안 추가/삭제/업데이트 불가능	-

특성	설명	예시
CONCURRENT	동기화 없이 여러 스레드에서 동시에 데이터 수정해도 안전	-
SUBSIZED	이 Spliterator와 분할되는 모든 Spliterator가 SIZED 특성을 가짐	-

characteristics() 메소드

```

List<Integer> list = List.of(1, 2, 3);
Spliterator<Integer> spliterator = list.spliterator();

// 특성 확인
int characteristics = spliterator.characteristics();
System.out.println(characteristics);
// ORDERED | SIZED | SUBSIZED |

```