병렬 스트림

소유자	종수 김
태그	

단일 스트림

병렬 스트림 준비 예제

MyLogger

```
package util;

import java.time.LocalTime;
import java.time.format.DateTimeFormatter;

public class MyLogger {
    private static final DateTimeFormatter formatter = DateTimeFormatter.ofPatt

    public static void log(Object obj) {
        String time = LocalTime.now().format(formatter);
        System.out.printf("%s [%9s] %s\n", time, Thread.currentThread().getNar
    }
}
```

- 현재 시간, 스레드 이름, 전달받은 객체를 로그로 출력

HeavyJob

```
package parallel;

import util.MyLogger;

public class HeavyJob {
    public static int heavyTask(int i) {
        MyLogger.log("calculate " + i + " → " + i * 10);
        try {
```

```
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        return i * 10;
    }

    public static int heavyTask(int i, String name) {
        MyLogger.log("[" + name + "] " + i + " → " + i * 10);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        return i * 10;
    }
}
```

- 오래 걸리는 작업을 시뮬레이션.

예제 1 - 단일 스트림

```
package parallel;

import util.MyLogger;

import java.util.stream.IntStream;

import static util.MyLogger.log;

public class ParallelMain1 {
    public static void main(String[] args) {
        long startTime = System.currentTimeMillis();

        int sum = IntStream.rangeClosed(1,8)
                .map(HeavyJob::heavyTask)
                .reduce(0,(a,b) → a + b);
```

```
        long endTime = System.currentTimeMillis();
        log("time: " + (endTime - startTime) + "ms, sum: " + sum);
    }
}
```

- map으로 1초씩 걸리는 작업을 8번 순차로 호출.
- sum을 통해서 값을 더해줌.
- 단일 스레드(Main 스레드)에서 작업을 순차적으로 수행하기 때문에 [main]쓰레드만 표시되며, 8초의 시간이 걸림.

스레드 직접 사용

앞서는 하나의 스레드로 1 ~ 8의 범위를 모두 계산.

여러 스레드를 동시에 사용해서 작업을 더 빨리 처리하는 것이 가능.

각 스레드는 한 번에 하나의 작업만 처리할 수 있으므로 1 ~ 8을 처리하는 큰 단위의 작업을 더 작은 단위의 작업 1 ~ 4 , 5 ~ 8과 같이 분할해야함.

```
package parallel;

import util.MyLogger;

import java.util.TreeMap;
import java.util.stream.IntStream;

import static util.MyLogger.log;

public class ParallelMain2 {
    public static void main(String[] args) throws InterruptedException {
        long startTime = System.currentTimeMillis();

        // 1. Fork - 작업을 분할.
        SumTask task1 = new SumTask(1, 4);
        SumTask task2 = new SumTask(5, 8);

        Thread thread1 = new Thread(task1, "thread-1");
        Thread thread2 = new Thread(task2, "thread-2");
```

```
        // 2. 분할 작업 처리
        thread1.start();
        thread2.start();

        // 3. join - 처리한 결과를 합침
        thread1.join();
        thread2.join();
        MyLogger.log("main 스레드 대기 완료");

        int sum = task1.result + task2.result;



        long endTime = System.currentTimeMillis();
        log("time: " + (endTime - startTime) + "ms, sum: " + sum);
    }
    static class SumTask implements Runnable {
        int startValue;
        int endValue;
        int result = 0;

        public SumTask(int startValue, int endValue) {
            this.startValue = startValue;
            this.endValue = endValue;
        }

        @Override
        public void run() {
            MyLogger.log("작업 시작");
            int sum = 0;
            for (int i = startValue; i <= endValue; i++) {
                int calculated = HeavyJob.heavyTask(i);
                sum += calculated;
            }
            result = sum;
            MyLogger.log("작업 완료 result = " + result);
        }
    }
```

```
}
```

- SumTask는 Runnable을 구현했고, 내부에서 1초씩 걸리는 heavyTask()를 루프 돌면서 합산.
- SumTask는 1 ~ 4 / 5 ~ 8로 작업을 두 개로 분할
- thread1.start(), thread2.start()로 각 스레드가 동시에 작업을 시작.
- thread1.join(), thread2.join()으로 두 스레드가 끝날 때 까지 main 쓰레드가 대기
- 작업 완료 후 task1 + task2로 sum

쓰레드 풀 사용

ExecutorService를 사용해서 병렬 처리 가능.

```
package parallel;

import util.MyLogger;

import java.util.concurrent.*;

import static util.MyLogger.log;

public class ParallelMain3 {
    public static void main(String[] args) throws InterruptedException, Executio

        // 쓰레드 풀 준비
        ExecutorService es = Executors.newFixedThreadPool(2);

        long startTime = System.currentTimeMillis();

        // 1. Fork - 작업을 분할.
        SumTask task1 = new SumTask(1, 4);
        SumTask task2 = new SumTask(5, 8);

        // 2. 분할 작업 처리
        Future<Integer> future1 = es.submit(task1);
        Future<Integer> future2 = es.submit(task2);
```

```
        // 3. join - 처리한 결과를 합침
        Integer result1 = future1.get();
        Integer result2 = future2.get();
        MyLogger.log("main 스레드 대기 완료");

        int sum = result1 + result2;


        long endTime = System.currentTimeMillis();
        log("time: " + (endTime - startTime) + "ms, sum: " + sum);

        es.shutdown();
    }
    static class SumTask implements Callable<Integer> {
        int startValue;
        int endValue;
        int result = 0;

        public SumTask(int startValue, int endValue) {
            this.startValue = startValue;
            this.endValue = endValue;
        }

        @Override
        public Integer call() throws Exception {
            MyLogger.log("작업 시작");
            int sum = 0;
            for (int i = startValue; i <= endValue; i++) {
                int calculated = HeavyJob.heavyTask(i);
                sum += calculated;
            }
            result = sum;
            MyLogger.log("작업 완료 result = " + result);
            return sum;
        }
    }
```

```
}
```

- `Executors.newFixedThreadPool(2)` 로 쓰레드 풀을 생성
  - 이 쓰레드 풀은 최대 2개의 쓰레드를 제공
  - 2개 이상의 작업이 실행 될 경우 그 작업은 앞선 작업이 끝나야 실행
- submit(Callable)로 쓰레드 풀에 작업을 맡기면 Future 객체를 반환 받음.
- 메인 쓰레드는 future.get()을 통해 실제 계산 결과가 반환될 때 까지 대기.

이 예제는 스레드 풀과 `Future`을 사용해서 결과값을 반환받는 방식으로 구현되었다. 작업이 완료되면 `Future`의 `get()` 메서드를 통해 결과를 얻는다. 참고로 `get()` 메서드는 블로킹 메서드이다. 이전 예제와 마찬가지로 2개의 스레드가 병렬로 계산을 처리하므로 약 4초가 소요된다.

실행 결과

```
15:31:19.238 [pool-1-thread-1] 작업 시작
15:31:19.238 [pool-1-thread-2] 작업 시작
15:31:19.244 [pool-1-thread-1] calculate 1 -> 10
15:31:19.244 [pool-1-thread-2] calculate 5 -> 50
15:31:20.247 [pool-1-thread-1] calculate 2 -> 20
15:31:20.247 [pool-1-thread-2] calculate 6 -> 60
15:31:21.248 [pool-1-thread-2] calculate 7 -> 70
15:31:21.253 [pool-1-thread-1] calculate 3 -> 30
15:31:22.252 [pool-1-thread-2] calculate 8 -> 80
15:31:22.258 [pool-1-thread-1] calculate 4 -> 40
15:31:23.258 [pool-1-thread-2] 작업 완료 result=260
15:31:23.258 [pool-1-thread-1] 작업 완료 result=100
15:31:23.270 [     main] time: 4040ms, sum: 360
```

- 이전 예제처럼 스레드가 2개이므로 각각 4개씩 나눠 처리한다.
- `Future`로 반환값을 쉽게 받아올 수 있기 때문에, 결과값을 합산하는 과정이 더 편리해졌다.
- 하지만 여전히 코드 레벨에서 분할/병합 로직을 직접 짜야 하고, 스레드 풀 생성과 관리도 개발자가 직접해야 한

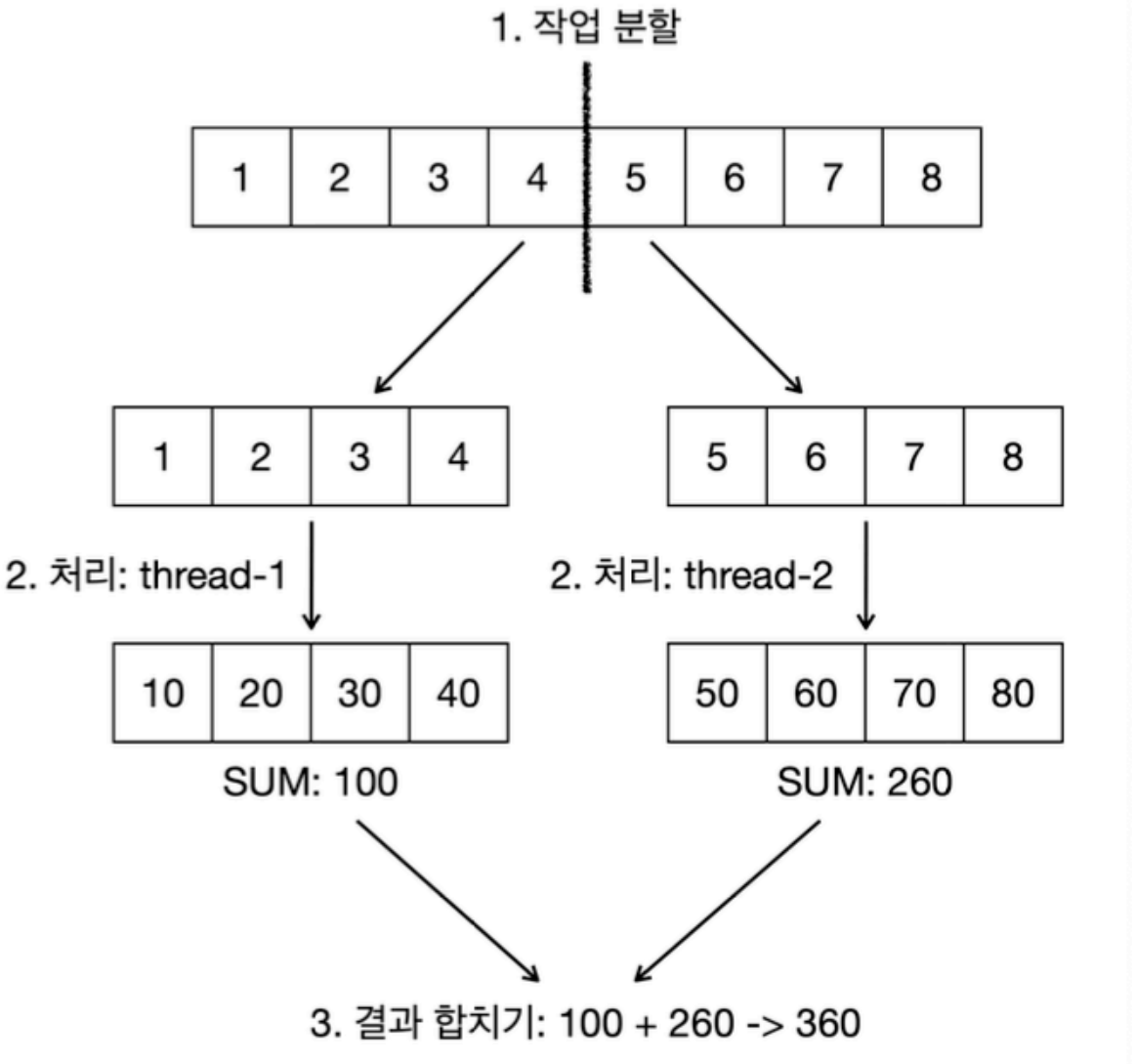
Fork / Join 패턴

분할(Fork), 처리(Execute), 모음(Join)

쓰레드는 한 번에 하나의 작업을 처리할 수 있음.
따라서, 하나의 큰 작업을 여러 쓰레드가 처리할 수 있는 작은 단위의 작업으로 분할(Fork)

이렇게, 분할한 작업을 각각의 쓰레드가 처리(Execute)
끝나면, 분할된 결과를 하나로 모아야함(Join)

분할 - 처리 - 모음의 단계로 이루어진 멀티쓰레딩 패턴을 Fork/Join 패턴이라고 하며,
병렬 프로그래밍에서 매우 효율적인 방식으로, 복잡한 작업을 병렬적으로 처리할 수 있게해 줌.



1. 작업 분할

`1 ~ 8` 분할

- `1 ~ 4`: `thread-1` 처리
- `5 ~ 8`: `thread-2` 처리

`1 ~ 8`의 작업을 절반으로 분할하자. 그래서 `1 ~ 4`의 작업은 `thread-1`이 처리하고, `5 ~ 8`의 작업은 `thread-2`가 처리하는 것이다. 이렇게 하면 작업의 수를 늘려서 여러 스레드가 동시에 많은 작업을 처리할 수 있다. 예제에서는 하나의 스레드가 처리하던 작업을 두 개의 스레드가 처리하므로 처리 속도를 최대 2배로 늘릴 수 있다.

이렇게 큰 작업을 여러 작은 작업으로 쪼개어(`Fork`) 각각의 스레드나 작업 단위로 할당하는 것을 포크(`Fork`)라 한다.
참고로 포크라는 이름은 식당에서 사용하는 포크가 여러 갈래로 나뉘어 있는 모양을 떠올려보면 된다. 이처럼 하나의 큰

작업을 여러 작은 작업으로 분할하는 것을 포크라 한다.

2. 처리

2. 처리(Execute)

- `1 ~ 4` 처리(`thread-1`)
  - 1 → 10
  - 2 → 20
  - 3 → 30
  - 4 → 40
  - 결과: 10 + 20 + 30 + 40 = 100
- `5 ~ 8` 처리(`thread-2`)
  - 5 → 50
  - 6 → 60
  - 7 → 70
  - 8 → 80
  - 결과: 50 + 60 + 70 + 80 = 260

`thread-1`, `thread-2`는 분할된 각각의 작업을 처리한다.

3. Join 모음, 결과 합치기

분할된 작업들이 모두 끝나면, 각 스레드 혹은 작업 단위별 결과를 **하나로 합쳐야한다.**
예제에서는 `thread1.join()`, `thread2.join()`을 통해 모든 스레드가 종료되길 기다린 뒤, `task1.result + task2.result`로 최종 결과를 계산한다.
Join은 이렇게 갈라진 작업들이 모두 끝난 뒤, 다시 **합류**하여 하나로 **결과를 모으는** 모습을 의미한다.

정리

이러한 분할 → 처리 → 모음의 과정을 더 편리하게 구현할 수 있는 방법으로 자바는 Fork / Join 프레임워크를 제공.

Fork/Join 프레임워크 - 소개

Fork / Join 프레임워크 소개

주요 개념

1. 분할 정복(Devide and Conquer) 전략
    - 큰 작업(task)을 작은 단위로 재귀적으로 분할(fork)
    - 각 작은 작업의 결과를 합쳐(join) 최종 결과를 생성
    - 멀티코어 환경에서 작업을 효율적으로 분산 처리
2. 작업 훔치기 알고리즘
    - 각 스레드는 자신의 작업 큐를 가짐
    - 작업이 없는 스레드는 다른 바쁜 스레드의 큐에서 작업을 “훔쳐와서” 대신 처리
    - 부하 균형을 자동으로 조절하여 효율성 향상

주요 클래스

ForkJoinPool

- Fork/Join 작업을 실행하는 특수한 ExecutorService 쓰레드 풀
- 작업 스케줄링 및 스레드 관리 담당
- 기본적으로 사용 가능한 프로세서 수 만큼 쓰레드 생성
    - Ex) CPU 코어가 10 코어면 10개의 쓰레드
- 분할 정복과 작업 훔치기에 특화된 쓰레드풀.

```
// 기본 풀 생성 (프로세서 수에 맞춰 스레드 생성)
```

```
ForkJoinPool pool = new ForkJoinPool();
// 특정 병렬 수준으로 풀 생성
ForkJoinPool customPool = new ForkJoinPool(4);
```

ForkJoinTask

- Fork / Join 작업의 기본 추상 클래스
- Future를 구현

RecursiveTask<V>

- 결과를 반환하는 작업

RecursiveAction

- 결과를 반환하지 않는 작업

RecursiveTask / RecursiveAction의 구현 방법

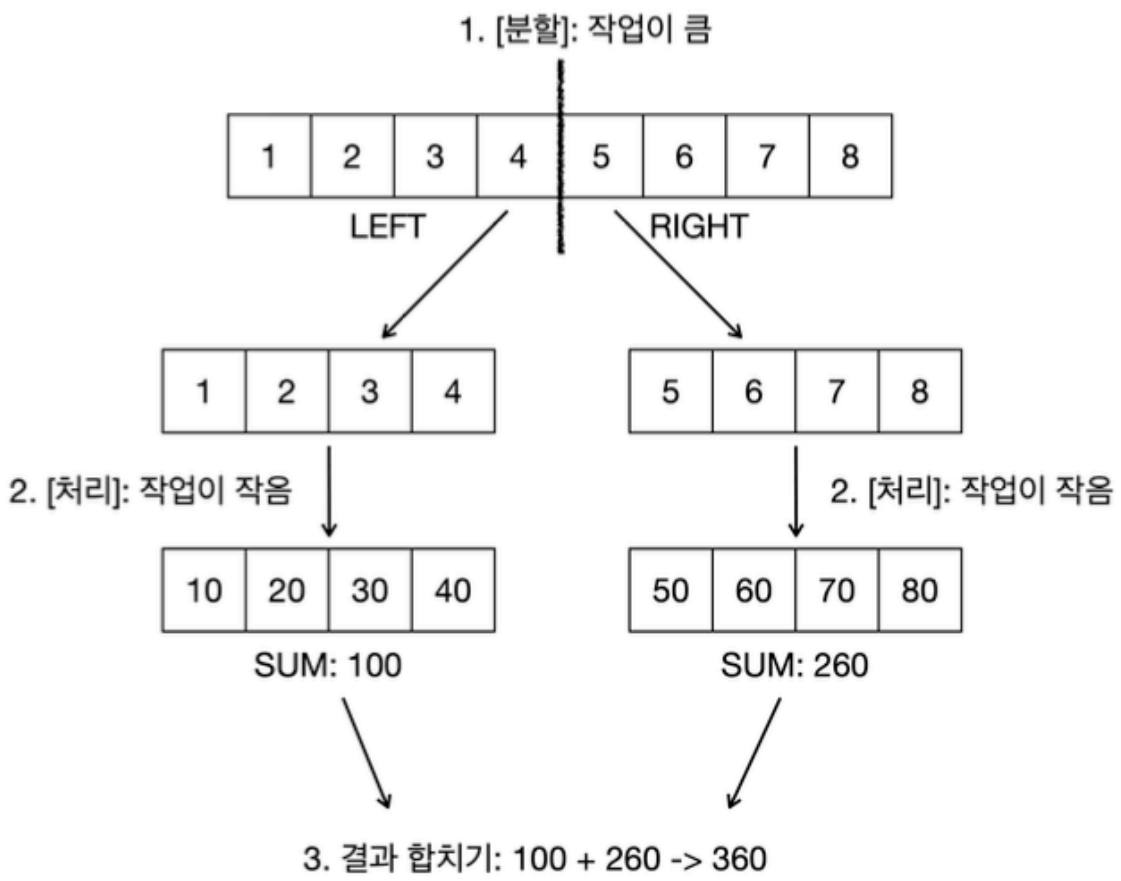
- `compute()` 메서드를 재정의해서 필요한 작업 로직을 작성한다.
- 일반적으로 일정 기준(임계값)을 두고, **작업 범위가 작으면 직접 처리**하고, **크면 작업을 둘로 분할**하여 각각 병렬로 처리하도록 구현한다.

fork() / join() 메서드

- `fork()` : 현재 스레드에서 다른 스레드로 작업을 **분할**하여 보내는 동작(비동기 실행)
- `join()` : 분할된 작업이 끝날 때까지 기다린 후 결과를 가져오는 동작

> **참고**: Fork/Join 프레임워크를 실무에서 직접적으로 다루는 일은 드물다. 따라서 이런게 있다 정도만 알아두고 넘어가자.
> 개념 정도만 대략 알아두면 충분하다.

Fork / Join 프레임워크 활용



핵심은 작업의 크기가 임계값 보다 크면 분할하고, 임계값 보다 같거나 작으면 직접 처리하는 것이다.

예를 들어 작업의 크기가 8이고, 임계값이 4라고 가정해보자.

1. **Fork**: 작업의 크기가 8이면 임계값을 넘었다. 따라서 작업을 절반으로 분할한다.
2. **Execute**: 다음으로 작업의 크기가 4라면 임계값의 범위 안에 들어오므로 작업을 분할하지 않고, 처리한다.
3. **Join**: 최종 결과를 합친다.

```
package parallel.forkjoin;

import parallel.HeavyJob;
import util.MyLogger;

import java.util.List;
import java.util.concurrent.RecursiveTask;

import static util.MyLogger.log;

public class SumTask extends RecursiveTask<Integer> {
    private static final int THRESHOLD = 4; // 임계 값
```

```
private final List<Integer> list;

public SumTask(List<Integer> list) {
    this.list = list;
}

@Override
protected Integer compute() {
    // 작업 범위가 적으면 직접 계산
    if (list.size() <= THRESHOLD) {
        log("[처리 시작] " + list);
        int sum = list.stream()
                .mapToInt(HeavyJob::heavyTask)
                .sum();
        log("[처리 완료] " + list + " → sum: " + sum);
        return sum;
    } else {
        // 작업 범위가 크면 반으로 나누어 병렬 처리
        int mid = list.size() / 2;
        List<Integer> leftList = list.subList(0, mid);
        List<Integer> rightList = list.subList(mid, list.size());
        log("[분할] " + list + " → LEFT" + leftList + ", RIGHT" + rightList);

        SumTask leftTask = new SumTask(leftList);
        SumTask rightTask = new SumTask(rightList);

        // 왼쪽 작업은 다른 쓰레드에서 처리
        leftTask.fork();
        // 오른쪽 작업은 현재 쓰레드에서 처리
                int rightResult = rightTask.compute(); // 현재 내 쓰레드에서 실행되.

        // 왼쪽 작업 결과를 기다림
        int leftResult = leftTask.join();

        // 왼쪽과 오른쪽 결과를 합침
        int joinSum = leftResult + rightResult;
        log("LEFT[" + leftResult + "] + RIGHT[" + rightResult + "] → sum: " + jo
```

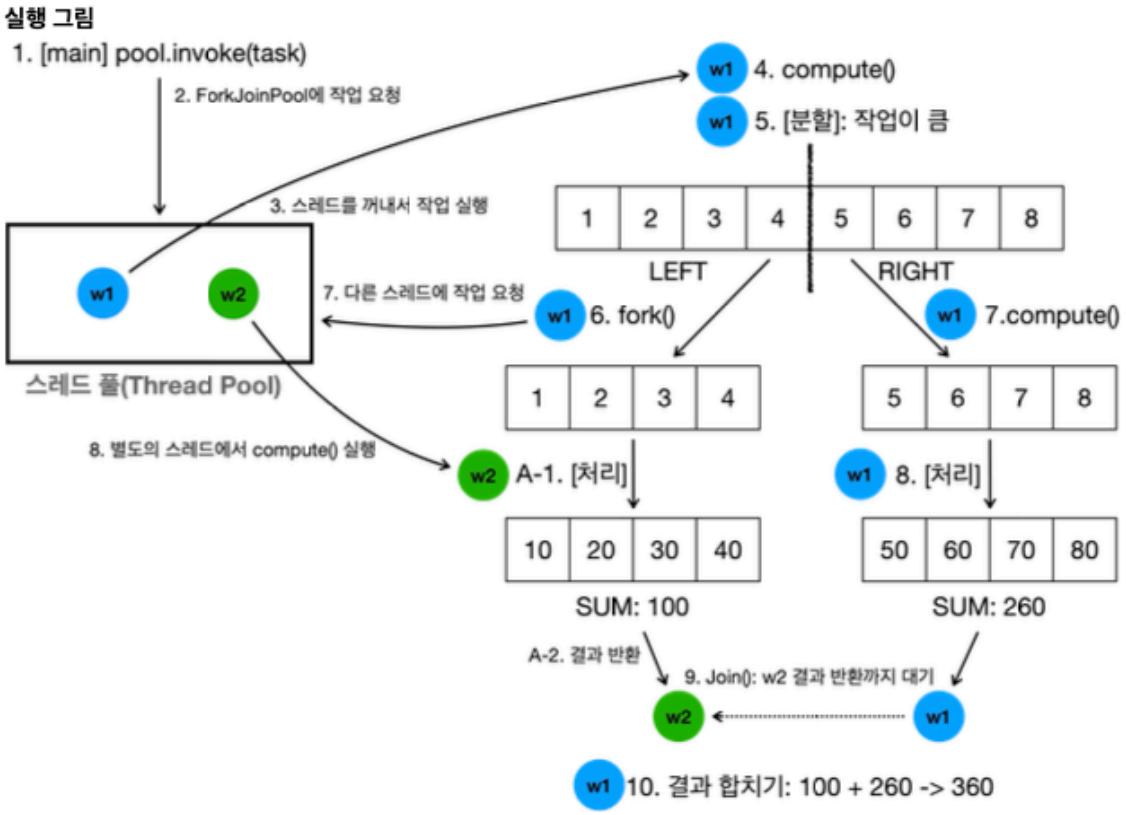
```
            return joinSum;
        }
    }
}
```

- THRESHOLD (임계값) : 작업을 더 이상 분할하지 않고, 직접 처리할 리스트의 크기를 정의한다. 여기서는 4로 설정.
- 작업 분할 : 리스트의 크기가 임계값보다 크면, 리스트를 반으로 나누어 leftList, rightList로 분할
- fork(), compute()
  - fork() : 왼쪽 작업을 다른 쓰레드에 위임하여 병렬 처리
  - compute() : 오른쪽 작업을 현재 쓰레드에서 직접 수행(재귀 호출)
- join() : 분할된 왼쪽 작업이 완료될 때 까지 기다린 후 결과를 가져옴
- 결과 합산 : 왼쪽과 오른쪽 결과를 합쳐 최종 결과를 반환

```
package parallel.forkjoin;

import java.util.List;
import java.util.concurrent.ForkJoinPool;
import java.util.stream.IntStream;

import static util.MyLogger.log;

public class ForkJoinMain1 {
    public static void main(String[] args) {
        List<Integer> data = IntStream.rangeClosed(1, 8)
                .boxed()
                .toList();
        log("[생성] " + data);

        // ForkJoinPool 생성 및 작업 수행

        ForkJoinPool pool = new ForkJoinPool(10);
        long startTime = System.currentTimeMillis();
        SumTask task = new SumTask(data);
```

```
        // 병렬로 합을 구한 후 결과 출력
        int result = pool.invoke(task);
        pool.shutdown();
        long endTime = System.currentTimeMillis();
        log("time: " + (endTime - startTime) + "ms, sum: " + result);
        log("pool: " + pool);
    }
}
```

1. 데이터 생성 : IntStream.rangeClosed(1,8
2. ForkJoinPool 생성
    a. new ForkJoinPool 로 최대 10개 스레드를 사용할 수 있는 풀을 생성
    b. 기본 생성자는 시스템의 프로세서 수에 맞춰 쓰레드 생성
3. invoke() : 메인 쓰레드가 pool.invoke(task)를 호출하면, SumTask를 쓰레드 풀에 전달.
   SumTask는 ForkJoinPool에 있는 별도의 쓰레드에서 실행. 메인쓰레드는 작업이 완료될 때 까지 기다린 후 결과를 받음.
4. pool.close() 더 이상 작업이 없으므로 풀 종료
5. 결과 출력



1. Main 쓰레드가 invoke(task)를 호출
2. ForkJoinPool에 작업을 요청
3. 쓰레드 풀은 쓰레드를 꺼내서 작업을 실행 여기서는 ForkJoinPool-1-worker-1 쓰레드가 실행.
    a. w1
4. w1 쓰레드는 task(SumTask)의 compute()를 호출.

작업 분할

```
[w1] [분할] [1, 2, 3, 4, 5, 6, 7, 8] -> LEFT[1, 2, 3, 4], RIGHT[5, 6, 7, 8]
```

5. 리스트 크기가 `THRESHOLD(4)` 보다 크므로 분할됨.

    - `[1,2,3,4,5,6,7,8]` 이 `LEFT[1,2,3,4]` 와 `RIGHT[5,6,7,8]` 로 나뉨.
6. `w1` 은 분할한 왼쪽 리스트인 `LEFT[1,2,3,4]` 는 `fork(leftTask)` 를 호출해서 다른 스레드가 작업을 처리하도록 요청함
7. `w1` 은 분할할 오른쪽 리스트인 `RIGHT[5,6,7,8]` 는 자기 자신의 메서드인 `compute(rightTask)` 를 호출해서 자기 자신이 스스로 처리함 (재귀 호출)

병렬 처리

- 각 스레드가 동시에 `HeavyJob.heavyTask()`를 실행하며 병렬로 계산.
    - 8. `w1` 스레드가 `[5,6,7,8]` 을 순서대로 처리, SUM: 260 (리스트 크기가 `THRESHOLD(4)` 이하)
    - A-1. `w2` 스레드가 `[1,2,3,4]` 를 순서대로 처리, SUM: 100 (리스트 크기가 `THRESHOLD(4)` 이하)
- `[1,2,3,4]` 작업의 합은 100, `[5,6,7,8]` 의 작업의 합은 260

작업 완료:

9. 최종 결과의 합을 구하기 위해 `w1` 스레드는 `w2` 스레드의 작업에 `join()` 메서드를 호출해서 `w2` 의 결과를 기다림
10. 두 결과가 합쳐져 최종 합계 360이 계산됨

정리

- Fork/Join 프레임워크를 사용하면 RecursiveTask를 통해 작업을 재귀적으로 분할. 단순히 2개로만 분할해서 쓰레드도 동시에 2개만 사용할 수 있었음.
- THRESHOLD(임계값)을 더 줄여서 작업을 더 잘게 분할하면, 더 많은 쓰레드를 활용할 수 있음. 물론 이 경우 풀의 쓰레드 수도 2개보다 더 많아야 효과가 있음.

Fork / Join 프레임워크 2 - 작업 훔치기

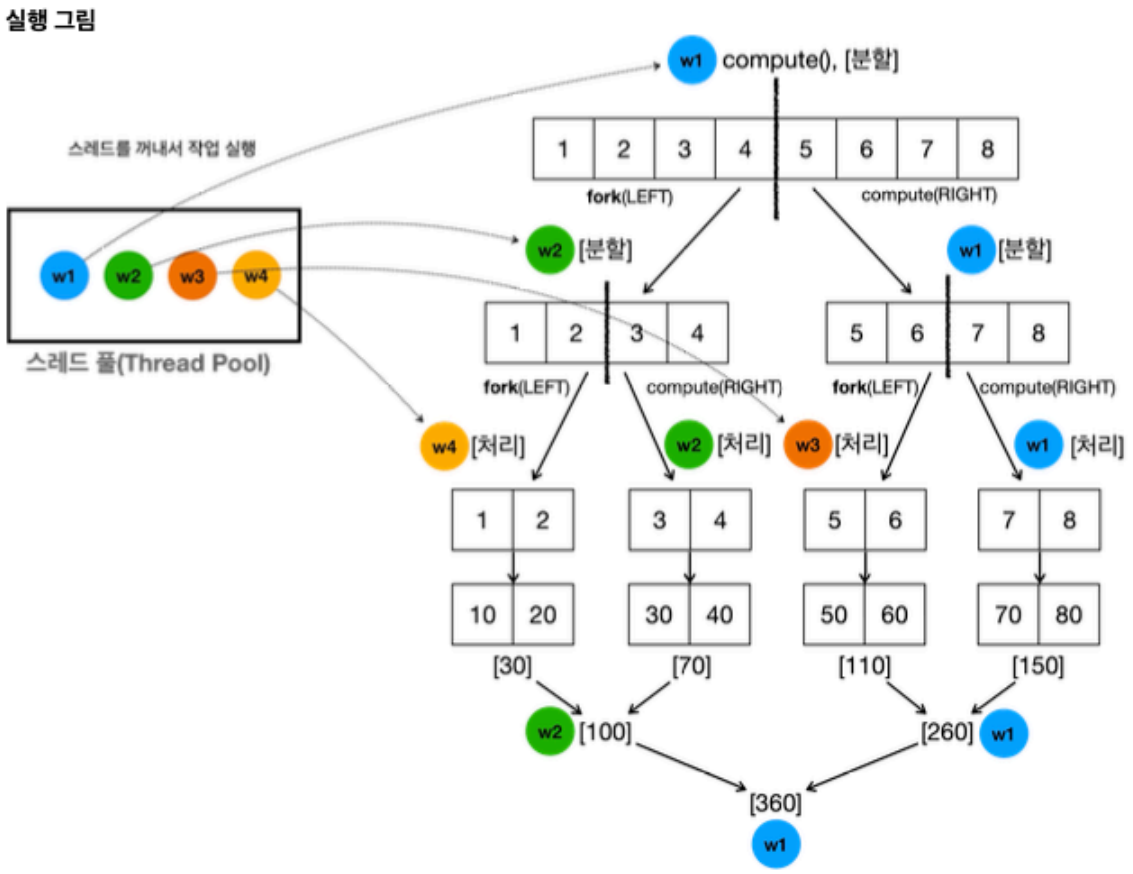
```
public class SumTask extends RecursiveTask<Integer> {
    //private static final int THRESHOLD = 4; // 임계값
    private static final int THRESHOLD = 2; // 임계값 변경
    ...
}
```

그리고 `ForkJoinMain1`을 실행해보자.

실행 결과

```
16:02:19.416 [     main] [생성] [1, 2, 3, 4, 5, 6, 7, 8]
16:02:19.425 [ForkJoinPool-1-worker-1] [분할] [1, 2, 3, 4, 5, 6, 7, 8] ->
LEFT[1, 2, 3, 4], RIGHT[5, 6, 7, 8]
16:02:19.425 [ForkJoinPool-1-worker-1] [분할] [5, 6, 7, 8] -> LEFT[5, 6],
RIGHT[7, 8]
16:02:19.425 [ForkJoinPool-1-worker-2] [분할] [1, 2, 3, 4] -> LEFT[1, 2],
RIGHT[3, 4]
16:02:19.425 [ForkJoinPool-1-worker-1] [처리 시작] [7, 8]
16:02:19.425 [ForkJoinPool-1-worker-2] [처리 시작] [3, 4]
16:02:19.425 [ForkJoinPool-1-worker-3] [처리 시작] [5, 6]
16:02:19.425 [ForkJoinPool-1-worker-4] [처리 시작] [1, 2]
16:02:19.430 [ForkJoinPool-1-worker-4] calculate 1 -> 10
16:02:19.430 [ForkJoinPool-1-worker-2] calculate 3 -> 30
16:02:19.430 [ForkJoinPool-1-worker-3] calculate 5 -> 50
16:02:19.430 [ForkJoinPool-1-worker-1] calculate 7 -> 70
16:02:20.435 [ForkJoinPool-1-worker-1] calculate 8 -> 80
16:02:20.435 [ForkJoinPool-1-worker-4] calculate 2 -> 20
16:02:20.435 [ForkJoinPool-1-worker-3] calculate 6 -> 60
16:02:20.435 [ForkJoinPool-1-worker-2] calculate 4 -> 40
16:02:21.447 [ForkJoinPool-1-worker-3] [처리 완료] [5, 6] -> sum: 110
16:02:21.447 [ForkJoinPool-1-worker-1] [처리 완료] [7, 8] -> sum: 150
16:02:21.448 [ForkJoinPool-1-worker-4] [처리 완료] [1, 2] -> sum: 30
16:02:21.448 [ForkJoinPool-1-worker-2] [처리 완료] [3, 4] -> sum: 70
16:02:21.454 [ForkJoinPool-1-worker-2] LEFT[30] + RIGHT[70] -> sum: 100
16:02:21.454 [ForkJoinPool-1-worker-1] LEFT[110] + RIGHT[150] -> sum: 260
16:02:21.455 [ForkJoinPool-1-worker-1] LEFT[100] + RIGHT[260] -> sum: 360
16:02:21.455 [     main] time: 2030ms, sum: 360
16:02:21.456 [     main] pool:
java.util.concurrent.ForkJoinPool@57baeedf[Terminated, parallelism = 10, size
= 0, active = 0, running = 0, steals = 4, tasks = 0, submissions = 0]
```

- 임계 값을 더 낮추어, 작업을 더 세분화



- 임계 값을 낮춤으로써 더 많은 쓰레드(총 4개)가 병렬로 작업을 처리.
- 이전 실행(임계값 4)에서는 2개의 쓰레드만 사용했음.
- 여러 쓰레드를 사용하여 작업을 더 빠르게 처리

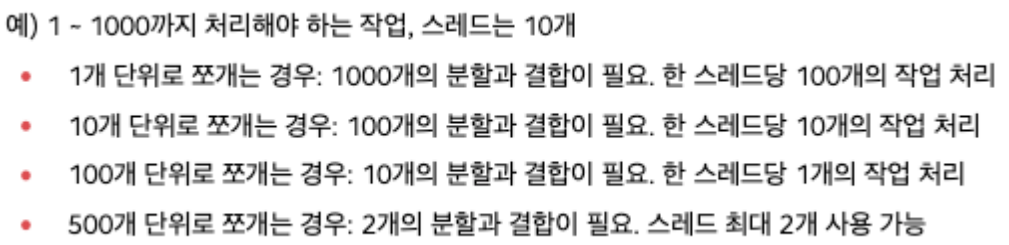
작업 훔치기 알고리즘

- Fork / Join 풀의 쓰레드는 각자 자신의 작업 큐를 가짐.
    - 덕분에 작업을 큐에서 가져가기 위한 스레드간 경합이 줄어듦.
- 그리고 자신의 작업이 없는 경우, 쓰레드가 할 일이 없는 경우에 다른 쓰레드의 작업 큐에 대기중인 작업을 훔쳐서 대신 처리.
- 추 후 필요하면, 다시 학습.

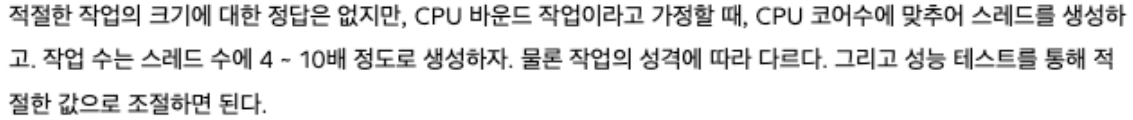
작업량이 불균형할 경우 작업 훔치기 알고리즘이 동작하여 유휴 스레드가 다른 바쁜 스레드의 작업을 가져와 처리함으로써 전체 효율성을 높일 수 있음.

적절한 작업 크기

- 너무 작은 단위로 작업을 분할하면, 스레드 생성과 관리에 드는 오버헤드가 커질 수 있음.
- 너무 큰 단위로 작업을 분할하면 병렬 처리의 이점을 충분히 활용하지 못할 수 있음.



- 적절한 작업의 크기에 대한 정답은 없으나, CPU 코어 수에 맞추어 해보고, 성능 테스트 이용
- 작업 시간이 완전히 동일하게 처리된다고 가정하면(이상적인 상태) 한 스레드당 1개의 작업을 처리하는 것이 좋음.
    - 스레드를 100% 사용하면서, 분할과 결합의 오버헤드를 최소화 할 수 있기 때문에.
- 하지만, 작업 시간이 다른 경우를 고려한다면 한 쓰레드 당 1개의 작업 보다는 더 잘게 쪼개어 두는게 좋음.
    - 작업 훔쳐가기 기능을 통해 먼저 끝난 쓰레드가 이미 작업이 남아있는 쓰레드의 작업을 훔쳐서 작업을 진행하기 때문.



Fork / Join 프레임워크 3 - 공용 풀

Fork / Join 공용 풀(Common Pool)

Fork / Join 작업을 위한 자바가 제공하는 기본 쓰레드 풀

```
// 자바 8 이상에서는 공용 풀(common pool) 사용 가능
ForkJoinPool commonPool = ForkJoinPool.commonPool();
```

Fork/Join 공용 풀의 특징

- **시스템 전체에서 공유**: 애플리케이션 내에서 **단일 인스턴스로 공유**되어 사용된다.
- **자동 생성**: 별도로 생성하지 않아도 `ForkJoinPool.commonPool()` 을 통해 접근할 수 있다.
- **편리한 사용**: 별도의 풀을 만들지 않고도 `RecursiveTask` / `RecursiveAction` 을 사용할 때 기본적으로 이 공용 풀이 사용된다.
- **병렬 스트림 활용**: 자바 8의 병렬 스트림은 내부적으로 이 공용 풀을 사용한다. (뒤에서 설명한다.)
- **자원 효율성**: 여러 곳에서 별도의 풀을 생성하는 대신 공용 풀을 사용함으로써 시스템 자원을 효율적으로 관리할 수 있다.
- **병렬 수준 자동 설정**: 기본적으로 시스템의 가용 프로세서 수에서 1을 뺀 값으로 병렬 수준(parallelism)이 설정된다. 예를 들어 CPU 코어가 14개라면 13개의 스레드가 사용된다. (자세한 이유는 뒤에서 설명)

```
package parallel.forkjoin;

import java.util.List;
import java.util.concurrent.ForkJoinPool;
import java.util.stream.IntStream;

import static util.MyLogger.log;

public class ForkJoinMain2 {
    public static void main(String[] args) {
        int processorCount = Runtime.getRuntime().availableProcessors();
        ForkJoinPool commonPool = ForkJoinPool.commonPool();
        log("processorCount = " + processorCount + ", commonPool = " + comm


        List<Integer> data = IntStream.rangeClosed(1, 8)
                .boxed()
                .toList();

        log("[생성] " + data);
        SumTask task = new SumTask(data);
        Integer result = task.invoke();// 공용 풀 사용
        log("최종 결과: " + result);
    }
}
```

- ForkJoinPool 인스턴스를 생성하지 않아도, 공용풀을 사용

공용 풀을 통한 실행

이전 예제는 ForkJoinPool을 생성한 다음에 pool.invoke(task)를 통해 풀에 직접 작업을 요청.

이번 예제를 보면 풀에 작업을 요청하는 것이 아니라 `task.invoke()`를 통해 작업(`RecursiveTask`)에 있는 `invoke()`를 직접 호출했다. 따라서 코드만 보면 풀을 전혀 사용하지 않는 것 처럼 보인다.

```
SumTask task = new SumTask(data);
Integer result = task.invoke(); // 공용 풀 사용
```

- 여기서 사용한 `invoke()` 메서드는 현재 스레드(여기서는 메인 스레드)에서 작업을 시작하지만, `fork()`로 작업 분할 후에는 공용 풀의 워커 스레드들이 분할된 작업을 처리한다.
  - 메인 스레드가 스레드 풀이 아닌 `RecursiveTask`의 `invoke()`를 직접 호출하면 메인 스레드가 작업의 `compute()`를 호출하게 된다. 이때 내부에서 `fork()`를 호출하면 공용 풀의 워커 스레드로 작업이 분할 된다.
- 메인 스레드는 최종 결과가 나올 때 까지 대기(블로킹)해야 한다. 따라서 그냥 대기하는 것 보다는 작업을 도와주는 편이 더 효율적이다.
  - `invoke()` : 호출 스레드가 작업을 도우면서 대기(블로킹)한다. 작업의 결과를 반환 받는다.
  - `fork()` : 작업을 비동기로 호출하려면 `invoke()` 대신에 `fork()`를 호출하면 된다. `Future`(`ForkJoinTask`)를 반환 받는다.

스레드 수

```
processorCount = 14, commonPool = 13
```

- `processorCount = 14` 현재 CPU의 코어 수 이다.
- `parallelism = 13` : 동시에 처리할 수 있는 작업 수준(스레드 수와 관련)
  - 현재 CPU 코어가 14개(`processorCount = 14`)이다. 따라서 공용 풀은 CPU - 1의 수 많큼 스레드를 생성한다.
  - 여기서는 최대 13개의 스레드를 생성해서 사용한다. (이유는 뒤에서 설명)

작업 실행 과정

- 메인 스레드와 워커 스레드들이 함께 작업을 처리한다.
- 워커 스레드 이름이 `worker-1`, `worker-2`, `worker-3`으로 표시된다(로깅 시 `ForkJoinPool.commonPool-` 접두사 생략)
- 메인 스레드도 작업 처리에 참여하는 것을 볼 수 있다(`[main]` 표시).

정리

- 공용 풀은 JVM이 종료될 때까지 계속 유지되므로, 별도로 풀을 종료(`shutdown()`)하지 않아도 된다.
- 이렇게 공용 풀(`ForkJoinPool.commonPool`)을 활용하면, 별도로 풀을 생성/관리하는 코드를 작성하지 않아도 간편하게 병렬 처리를 구현할 수 있다.

공용 풀 vs 커스텀 풀

이전 예제에서는 다음과 같이 커스텀 Fork/Join 풀을 생성했다.

```
ForkJoinPool pool = new ForkJoinPool();
Integer result = pool.invoke(task);
```

반면 이번 예제에서는 공용 풀을 사용했다.

```
Integer result = task.invoke();
```

차이점

1. **자원 관리**: 커스텀 풀은 명시적으로 생성하고 관리해야 하지만, 공용 풀은 시스템에서 자동으로 관리된다.
2. **재사용성**: 공용 풀은 여러 곳에서 공유할 수 있어 자원을 효율적으로 사용할 수 있다.
3. **설정 제어**: 커스텀 풀은 병렬 수준(스레드의 숫자), 스레드 팩토리 등을 세부적으로 제어할 수 있지만, 공용 풀은 기본 설정을 사용한다.
4. **라이프사이클**: 커스텀 풀은 명시적으로 종료해야 하지만, 공용 풀은 JVM이 관리한다. 따라서 종료하지 않아도 된다.

공용 풀이 CPU - 1 만큼 스레드를 생성하는 이유

기본적으로 자바의 Fork/Join 공용 풀은 시스템의 가용 CPU 코어 수 (`Runtime.getRuntime().availableProcessors()`)에서 1을 뺀 값을 병렬 수준(parallelism)으로 사용한다. 예를 들어 CPU가 14코어라면 공용 풀은 최대 13개의 워커 스레드를 생성한다. 그 이유는 다음과 같다.

메인 스레드의 참여
Fork/Join 작업은 공용 풀의 워커 스레드뿐만 아니라 메인 스레드도 연산에 참여할 수 있다. 메인 스레드가 단순히 대

기하지 않고 직접 작업을 도와주기 때문에, 공용 풀에서 스레드를 14개까지 만들 필요 없이 13개의 워커 스레드 + 1개의 메인 스레드로 충분히 CPU 코어를 활용할 수 있다.

다른 프로세스와의 자원 경쟁 고려
애플리케이션이 실행되는 환경에서는 OS나 다른 애플리케이션, 혹은 GC(가비지 컬렉션) 같은 내부 작업들도 CPU를 사용해야 한다. 모든 코어를 최대치로 점유하도록 설정하면 다른 중요한 작업이 지연되거나, 컨텍스트 스위칭 비용(context switching)이 증가할 수 있다. 따라서 하나의 코어를 여유분으로 남겨 두어 전체 시스템 성능을 보다 안정적으로 유지하려는 목적이 있다.

효율적인 자원 활용
일반적으로는 CPU 코어 수와 동일하게 스레드를 만들더라도 성능상 큰 문제는 없지만, 공용 풀에서 CPU 코어 수 - 1을 기본값으로 설정함으로써, 특정 상황(다른 작업 스레드나 OS 레벨 작업)에서도 병목을 일으키지 않는 선에서 효율적으로 CPU를 활용할 수 있다.

자바 병렬 스트림

Fork / Join 공용 풀을 사용해서 병렬 연산을 수행.

```
package parallel;

import java.util.concurrent.ForkJoinPool;
import java.util.stream.IntStream;

import static util.MyLogger.log;

public class ParallelMain4 {
    public static void main(String[] args) {
        int processorCount = Runtime.getRuntime().availableProcessors();
```

```
        ForkJoinPool commonPool = ForkJoinPool.commonPool();
        log("processorCount = " + processorCount + ", commonPool = " + comm

        long startTime = System.currentTimeMillis();

        int sum = IntStream.rangeClosed(1,8)
                .parallel()
                .map(HeavyJob::heavyTask)
                .reduce(0,(a,b) -> a + b);

        long endTime = System.currentTimeMillis();
        log("time: " + (endTime - startTime) + "ms, sum: " + sum);
    }
}
```

- parallel() 함수만 추가.
- 여러 쓰레드가 병렬로 해당 업무를 처리
- 가지고 있는 Fork/Join 공용 풀 개수만큼 쓰레드로 병렬 작업 처리.
- 직접 쓰레드를 만들 필요 없이 스트림에 parallel() 메서드만으로, 스트림이 자동으로 병렬 처리

스트림을 병렬로 처리하고 싶다고 parallel()로 '선언'만 하면, 실제 '어떻게' 할지는 자바 스트림이 내부적으로 알아서 처리.

병렬 스트림 사용시 주의점 - 1

Fork / Join 공용 풀은, CPU 바운드 작업(계산 집약적인 작업)을 위해서 설계되었음. 즉, 쓰레드가 주로 대기해야하는 I/O 바운드 작업에는 적합하ㅏ지 않음.

- I/O 바운드 작업은 주로 네트워크 호출을 통한 대기가 발생. Ex) 외부 API 호출, 데이터베이스 조회

주의사항 - Fork/Join 프레임워크는 CPU 바운드 작업에만 사용해라!

Fork/Join 프레임워크는 주로 CPU 바운드 작업(계산 집약적인 작업)을 처리하기 위해 설계되었다. 이러한 작업은 CPU 사용률이 높고 I/O 대기 시간이 적다. CPU 바운드 작업의 경우, 물리적인 CPU 코어와 비슷한 수의 스레드를 사용하는 것이 최적의 성능을 발휘할 수 있다. 스레드 수가 코어 수보다 많아지면 컨텍스트 스위칭 비용이 증가하고, 스레드 간 경쟁으로 인해 오히려 성능이 저하될 수 있기 때문이다.

I/O 작업처럼 블로킹 대기 시간이 긴 작업을 ForkJoinPool에서 처리하면 문제가 발생.

1. 쓰레드 블로킹에 따른 CPU 낭비
    - `ForkJoinPool`은 CPU 코어 수에 맞춰 제한된 개수의 스레드를 사용한다. (특히 공용 풀)
    - I/O 작업으로 스레드가 블로킹되면 CPU가 놀게 되어, 전체 병렬 처리 효율이 크게 떨어진다.
2. 컨텍스트 스위칭 오버헤드 증가
    - I/O 작업 때문에 스레드를 늘리면, 실제 연산보다 대기 시간이 길어지는 상황이 발생할 수 있다.
    - 스레드가 많아질수록 컨텍스트 스위칭(context switching) 비용도 증가하여 오히려 성능이 떨어질 수 있다.
3. 작업 훔치기 기법 무력화
    - `ForkJoinPool`이 제공하는 작업 훔치기 알고리즘은, CPU 바운드 작업에서 빠르게 작업 단위를 계속 처리하도록 설계되었다. (작업을 훔쳐서 쉬는 스레드 없이 계속 작업)

    - I/O 대기 시간이 많은 작업은 스래드가 I/O로 인해 대기하고 있는 경우가 많아, 작업 훔치기가 빛을 발휘하기 어렵고, 결과적으로 병렬 처리의 장점을 살리기 어렵다.
4. 분할 - 정복(작업 문할)이점 감소
    - Fork/Join 방식을 통해 작업을 잘게 나누어도, I/O 병목이 발생하면 CPU 병렬화 이점이 크게 줄어든다.
    - 오히려 분할된 작업들이 각기 I/O 대기를 반복하면서, `fork()`, `join()`에 따른 오버헤드만 증가할 수 있다.

- 즉, I/O로 인해 대기하는 경우에는 ForkJoinPool도 작업을 못하고 대기하게 되는 것.

정리

CPU 바운드 작업이라면 ForkJoinPool을 통해 병렬 계산을 극대화 가능,

I/O 바운드 작업은 별도의 전용 쓰레드 풀을 사용하는 편이 더 적합.
Ex) Executors.newFixedThreadPool() 등

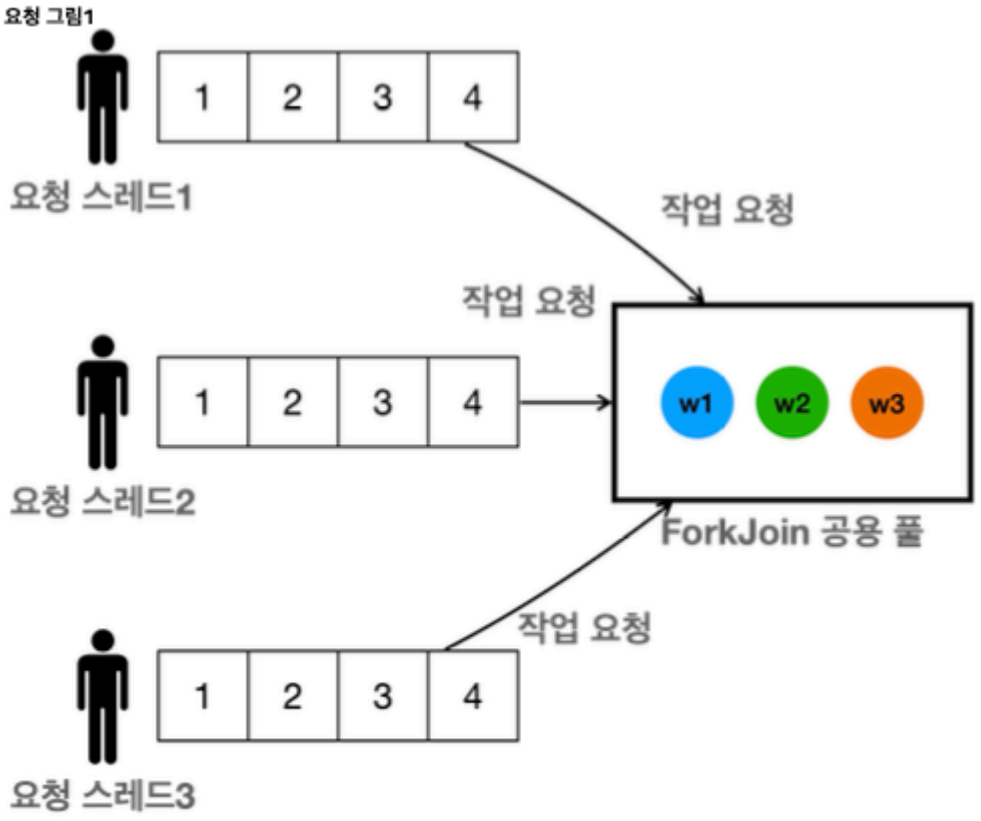
예제 5

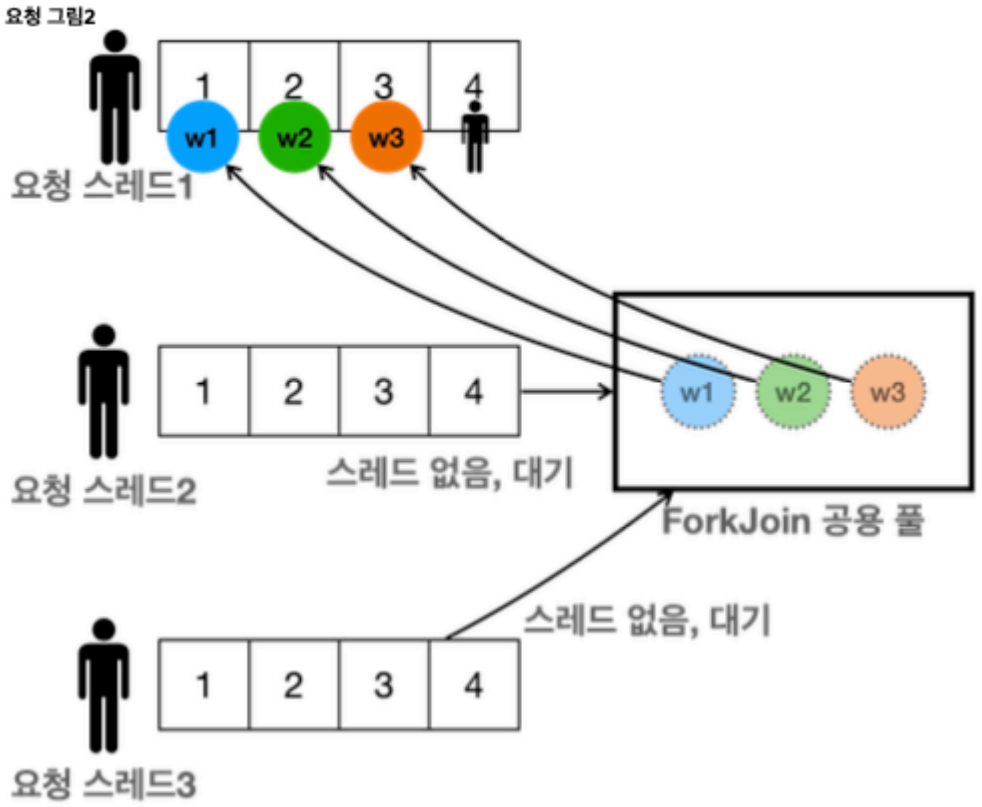
1. 여러 사용자가 동시에 서버를 호출하는 상황
2. 각 요청은 병렬 스트림을 사용하여 몇 가지 무거운 작업을 처리
3. 모든 요청이 동일한 공용 풀(ForkJoinPool)을 공유

```
package parallel;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.ForkJoinPool;
import java.util.stream.IntStream;

import static util.MyLogger.log;

public class ParallelMain5 {
    public static void main(String[] args) throws InterruptedException {
        //병렬 수준을 3으로 제한
        System.setProperty("java.util.concurrent.ForkJoinPool.common.parallelis
        System.out.println("ForkJoinPool.commonPool() = " + ForkJoinPool.comm

        // 요청 풀 추가
        ExecutorService requestPool = Executors.newFixedThreadPool(100);
        int nThreads = 20;
        for(int i = 0; i <= nThreads; i++) {
            String requestName = "request" + i;
            requestPool.submit(() -> logic(requestName));
            Thread.sleep(100);
        }
        requestPool.shutdown();

    }

    private static void logic(String requestName) {
        log("[" + requestName + "] START");
        long startTime = System.currentTimeMillis();
        int sum = IntStream.rangeClosed(1, 4)
                .parallel()
                .map(i -> HeavyJob.heavyTask(i, requestName))
```

```
                .reduce(0, (a, b) -> a + b);
        long endTime = System.currentTimeMillis();

        log("[" + requestName + "] time: " + (endTime - startTime) + "ms, sum: "
    }
}
```

- 코어가 4개라고 가정하고, 공용 풀의 병렬 수준을 3으로 제한.





- 공용 풀의 제한된 병렬성
  - 공용 풀은 병렬 수준이 3으로 설정되어 있어, 최대 3개의 작업만 동시에 처리할 수 있음.
    여기에, 요청 쓰레드도 자신의 작업에 참여하므로 각 작업당 총 4개의 쓰레드만 사용.
  - 따라서 총 12개의 요청(각각 4개의 작업)을 처리하는데 필요한 쓰레드 자원이 부족.
- 처리 시간의 불균형
  - 첫 번째 요청은 거의 모든 공용 풀 워커를 사용할 수 있었지만, 이후 요청들은 제한된 공용 풀 자원을 두고 경쟁해야 함. 따라서 완료 시간이 점점 느려짐.

    - request1: 1012ms (약 1초)
    - request2: 1931ms (약 2초)
    - request3: 2836ms (약 3초)

- 스레드 작업 분배
  - 일부 작업은 요청 쓰레드에서 직접 처리되고, 일부는 공용 풀에서 처리

- 요청 쓰레드가 작업을 도와주지만, 공용 풀의 쓰레드가 매우 부족하기 때문에 한계가 있음.

요청이 증가할수록 문제가 더 심각해짐.

핵심 문제점

1. 공용 풀 병목 현상
    a. 모든 병렬 스트림이 동일한 공용 풀을 공유하므로, 요청이 많아질수록 병목 현상이 발생
2. 자원 경쟁
    a. 여러 요청이 제한된 스레드 풀을 두고 경쟁하면서 요청의 성능이 저하
3. 예측 불가능한 성능
    a. 같은 작업이라도 동시에 실행되는, 다른 작업의 수에 따라 처리 시간이 크게 달라짐.

heavyTask()는 1초간 쓰레드가 대기하는 작업으로, I/O 바운드 작업에 가까움. - parallel() 제거하는게 더 처리가 빠름.
이런 종류의 작업은 Fork/Join 공용 풀 보다는 별도의 풀을 사용하는게 좋음.

> **주의! 실무에서 공용 풀은 절대! I/O 바운드 작업을 하면 안된다!**
> 실무에서 공용 풀에 I/O 바운드 작업을 해서 장애가 나는 경우가 있다.
> CPU 코어가 4개라면 공용 풀은 3개의 스레드만 사용한다. 그리고 공용 풀을 애플리케이션 전체에서 사용된다.
> 공용 풀에 있는 스레드 3개가 I/O 바운드 작업으로 대기하는 순간, 애플리케이션에서 공용 풀을 사용하는 모든 요청이 다 밀리게 된다.
> 예를 들어 공용 풀을 통해 외부 API를 호출하거나 데이터베이스를 호출하고 기다리는 경우가 있다. 만약 외부 API나 데이터베이스의 응답이 늦게 온다면 공용 풀의 3개의 스레드가 모두 I/O 응답을 대기하게 된다. 그리고 나머지 모든 요청이 공용 풀의 스레드를 기다리며 다 밀리게 되는 무시무시한 일이 발생한다.
>
> **공용 풀은 반드시 CPU 바운드(계산 집약적인)** 작업에만 사용해야 한다!
> 병렬 스트림은 처음부터 Fork/Join 공용 풀을 사용해서 CPU 바운드 작업에 맞도록 설계되어 있다.
> 따라서 이런 부분을 잘 모르고 실무에서 병렬 스트림에 I/O 대기 작업을 하는 것은 아주 위험한 일이다.
> 특히 병렬 스트림의 경우 단순히 `parallel()` 한 줄만 추가하면 병렬 처리가 되기 때문에, 어떤 스레드가 사용되는지도 제대로 이해하지 못하고 사용하는 경우가 있다. 병렬 스트림은 반드시 CPU 바운드 작업에만 사용하자!

그렇다면 여러 작업을 병렬로 처리해야 하는데, I/O 바운드 작업이 많을 때는 어떻게 하면 좋을까? 이때는 스레드를 직접 사용하거나, `ExecutorService` 등을 통해 별도의 스레드 풀을 사용해야 한다.

병렬 스트림 사용시 주의점 - 2

별도의 풀 사용

별도의 전용 쓰레드 풀을 사용

```
package parallel;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.Future;
import java.util.stream.IntStream;

import static util.MyLogger.log;

public class ParallelMain6 {
    public static void main(String[] args) throws InterruptedException {
        ExecutorService requestPool = Executors.newFixedThreadPool(100);

        // logic 처리 전용 쓰레드 풀 추가
        ExecutorService logicPool = Executors.newFixedThreadPool(400);

        int nThreads = 30;
        for(int i = 0; i <= nThreads; i++) {
            String requestName = "request" + i;
            requestPool.submit(() -> logic(requestName, logicPool));
            Thread.sleep(401);
        }
        requestPool.shutdown();
        logicPool.shutdown();
    }

    private static void logic(String requestName, ExecutorService es) {
        log("[" + requestName + "] START");
        long startTime = System.currentTimeMillis();

        Future<Integer> f1 = es.submit(() -> HeavyJob.heavyTask(1, requestNam
        Future<Integer> f2 = es.submit(() -> HeavyJob.heavyTask(2, requestNam
        Future<Integer> f3 = es.submit(() -> HeavyJob.heavyTask(3, requestNam
        Future<Integer> f4 = es.submit(() -> HeavyJob.heavyTask(4, requestNam
```

```
        // Future 결과 취합
        int sum;
        try {
            Integer v1 = f1.get();
            Integer v2 = f2.get();
            Integer v3 = f3.get();
            Integer v4 = f4.get();
            sum = v1 + v2 + v3 + v4;
        } catch (Exception e) {
            throw new RuntimeException(e);
        }

        long endTime = System.currentTimeMillis();

        log("[" + requestName + "] time: " + (endTime - startTime) + "ms, sum: "
    }
}
```

- 전용 로직 풀 추가
  - 최대 400개의 스레드를 가진 별도의 풀을 생성해서 병렬 작업 처리에 사용.
- 병렬 스트림 대신 커스텀 스레드 풀 사용
  - 병렬 스트림(parallel)을 사용하지 않고, 직접 전용 스레드 풀에 작업 제출
- 결과 취합 방식
  - Future.get()을 사용하여 각 작업의 결과를 기다렸다가 취합.

- **일관된 처리 시간**
  - 모든 요청이 약 1초 내외로 처리되었다.
  - 이전 예제에서 관찰된 요청별 처리 시간이 지연되는 문제가 해결되었다.
- **독립적인 스레드 할당**
  - 각 요청의 작업들이 전용 풀(pool-2-thread-N)에서 처리된다.
  - Fork/Join 공용 풀 스레드(ForkJoinPool.commonPool-worker-N)가 사용되지 않았다.
- **확장성 향상**
  - 400개의 스레드를 가진 풀을 사용함으로써, 동시에 여러 요청을 효율적으로 처리한다.
  - 공용 풀 병목 현상이 발생하지 않았다.

이 예제는 작업 유형에 적합한 전용 스레드 풀을 사용하는 것의 이점을 보여준다. 특히 I/O 바운드 작업이나 많은 동시 요청을 처리하는 서버 환경에서는 이러한 접근 방식이 더 효과적이다.

코드 개선

```
package parallel;

import java.util.List;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.stream.IntStream;

import static util.MyLogger.log;

public class ParallelMain7 {
    public static void main(String[] args) throws InterruptedException {
        ExecutorService requestPool = Executors.newFixedThreadPool(100);

        // logic 처리 전용 쓰레드 풀 추가
        ExecutorService logicPool = Executors.newFixedThreadPool(400);

        int nThreads = 30;
        for(int i = 0; i <= nThreads; i++) {
            String requestName = "request" + i;
            requestPool.submit(() -> logic(requestName, logicPool));
            Thread.sleep(100);
```

```
        }
        requestPool.shutdown();
        logicPool.shutdown();
    }

    private static void logic(String requestName, ExecutorService es) {
        log("[" + requestName + "] START");
        long startTime = System.currentTimeMillis();

        List<Future<Integer>> futures = IntStream.range(1, 4)
                .mapToObj(i -> es.submit(() -> HeavyJob.heavyTask(i, requestName)))
                .toList();

        int sum = futures.stream()
                .mapToInt(f -> {
                    try {
                        return f.get();
                    } catch (InterruptedException e) {
                        throw new RuntimeException(e);
                    } catch (ExecutionException e) {
                        throw new RuntimeException(e);
                    }
                }).sum();

        long endTime = System.currentTimeMillis();

        log("[" + requestName + "] time: " + (endTime - startTime) + "ms, sum: "
    }
}
```

- 스트림 API 활용
- 코드 간결화.

정리

1. 단일 스트림 vs 멀티 스레드

a. 단일 스트림 : 코드가 간단하지만, 한 번에 하나의 스레드만 실행되어 시간이 오래걸림

b. 멀티 스레드 : 여러 스레드를 직접 생성해 병렬로 작업을 처리할 수 있으나, 스레드 생성, 관리, 예외 처리 등이 복잡.

2. **스레드 풀(ExecutorService)**
   - 스레드를 직접 생성, 제어하는 대신, 자바가 제공하는 스레드 풀을 활용해 멀티스레드를 더 쉽게 사용할 수 있다.
   - `submit(Callable)`과 `Future`를 통해 작업을 비동기로 처리하고, 결과를 손쉽게 받아올 수 있다.

3. **Fork/Join 패턴과 Fork/Join 프레임워크**
   - 큰 작업을 잘게 **분할(Fork)** 한 뒤 여러 스레드가 **병렬로 처리**하고, 최종 결과를 **합치기(Join)** 하는 전형적인 병렬 처리 패턴이다.
   - 자바의 Fork/Join 프레임워크(`ForkJoinPool`, `RecursiveTask`, `RecursiveAction`)는 이러한 패턴을 편리하게 지원한다.
   - **작업 훔치기(Work-Stealing)** 알고리즘을 통해 각 스레드가 할당받은 작업이 없으면, 다른 스레드의 큐에 있는 작업을 훔쳐서 효율적으로 분산 처리한다.
   - **CPU 바운드** 작업(계산 집약적)일 때 최적의 효과를 낸다.

4. **자바 병렬 스트림**
   - 람다 스트림에서 `parallel()` 한 줄만 추가해도 내부적으로 Fork/Join 공용 풀을 사용하여 자동으로 병렬 처리한다.
   - 개발자는 복잡한 멀티스레드 코드를 짤 필요 없이 선언적으로 병렬 연산을 수행할 수 있다.
   - **단점**: 공용 풀을 공유하므로, I/O 대기 작업이나 동시 요청이 많아지는 상황에서 병목 현상이 발생할 수 있다.

5. 병렬 스트림 사용 시 주의 사항

   a. CPU 바운드 (계산 집약적) 작업에만 사용하는 것이 권장.

   b. I/O 바운드 작업(DB 조회, 외부 API 호출 등)은 오랜 대기 시간이 발생하므로, 제한된 스레드만 쓰는 Fork / Join 공용 풀과 궁합이 좋지 않음.

   c. 서버 환경에서 여러 요청이 동시에 병렬 스트림을 사용하면 공용 풀이 빠르게 포화되어 전체 성능이 저하될 수 있음.

6. **별도의 풀 사용**
    - I/O 바운드 작업처럼 대기가 긴 경우에는 **전용 스레드 풀(ExecutorService)**을 만들어 사용하는 것을 권장한다.
    - 스레드 풀의 크기, 스레드 생성 정책, 큐 타입 등을 상황에 맞게 튜닝할 수 있어 확장성과 안정성이 높아진다.

결론

- 자바 병렬 스트림은 CPU 바운드 작업을 빠르고 쉽게 병렬 처리하기에는 좋지만, 공용 풀을 이용한다는 점에서 I/O 바운드나 동시 요청이 많은 시스템에는 주의가 필요하다.
- 병렬 스트림만으로 모든 병렬 처리를 해결하려 하기보다는, **작업 특성**(CPU 바운드 vs I/O 바운드)과 **시스템 환경**(서버 동시 요청, 리소스 상황)을 고려하여 **Fork/Join** 프레임워크 또는 **별도의 스레드 풀**을 적절히 조합해 사용하는 것이 중요하다.

CompletableFuture와 주의 사항

CompletableFuture를 생성할 때는 별도의 쓰레드 풀을 반드시 지정해야 함.

그렇지 않으면 Fork / Join 공용 풀이 대신 사용됨.

CompletableFuture를 사용할 때는 **'반드시 커스텀 풀을 지정해서 사용'**

```
package parallel.forkjoin;

import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

import static util.MyLogger.log;

public class CompletableFutureMain {
    public static void main(String[] args) {
        CompletableFuture.runAsync(() -> log("Fork/Join")); // Fork / Join 공용 풀

        ExecutorService es = Executors.newFixedThreadPool(100);
        CompletableFuture.runAsync(() -> log("Custom Pool"), es);
        es.shutdown();
    }
}
```

```
21:55:42.763 [pool-1-thread-1] Custom Pool
21:55:42.763 [ForkJoinPool.commonPool-worker-1] Fork/Join
```

- 풀을 지정하지 않을 시 공용 풀을 사용함.
- 공용 풀은 코어 -1개 !
- 즉, I/O 작업이 많아서 블로킹 걸리는 경우, 코어 수만으로는 요청자 수를 감당할 수가 없음.