뒤태지존의 끄적거림

Java Concurrency Evolution

Dec 11, 2020 in Language

Java Concurrency Evolution

DZone에서 본 글인데 동시성 처리 관련 여러 방식을 비교하면서, 프로젝트 룸(Loom) 코드도 구경할 수 있어서 원글 작성자의 허락을 받고 우리말로 옮겨본다.

> 원문: https://dzone.com/articles/java-concurrency-evolution

자바는 초창기부터 스레드를 사용해서 동시성 프로그래밍을 할 수 있었다. 자바 1.1 버전까지는 JVM에서 그린 스레드(가상 스레드)를 지원했지만 그 이후 버전에서는 폐기되고 OS 네이티브 스레드를 사용하게 됐다. 하지만 오랫동안 관 속에 묻혀있던 가상 스레드를 다시 부활시키는 프로젝트 룸(Loom)이 수면 위로 부상하면서, 폐기됐던 가상 스레드가 다시 부활하여 주류에 올라설 수 있게 됐다.

이 글의 목적은 자바의 스레드/동시성 처리 진화 과정에 있었던 주요 마일스톤을 살펴보는 것이다. 스레드와 동시성을 다루는 자료는 차고 넘쳐나므로 이 글의 목적에서 벗어나는 다음 내용은 여기에서는 다루지 않는다.

- 에러 처리: 거의 다루지 않으며 코드 가독성을 높이기 위해 Lombok의 `@SneakyThrows`만 사용한다.
- 동시성 라이브러리나 프레임워크: JVM에는 Quasar, Akka, Guava, EA Async 등 동시성을 다루는 라이브러리와 프레임워크가 아주 많다.
- 복잡한 종료 조건: 주어진 태스크(task)를 수행하는데 필요한 스레드를 시작하고 나서, 스레드 종료까지 얼마나 기다려야 하는지 항상 명확하지는 않다.
- 스레드 간 동기화: 이 얘기를 다뤘다간 글을 끝맺을 수 없을 것만 같다.

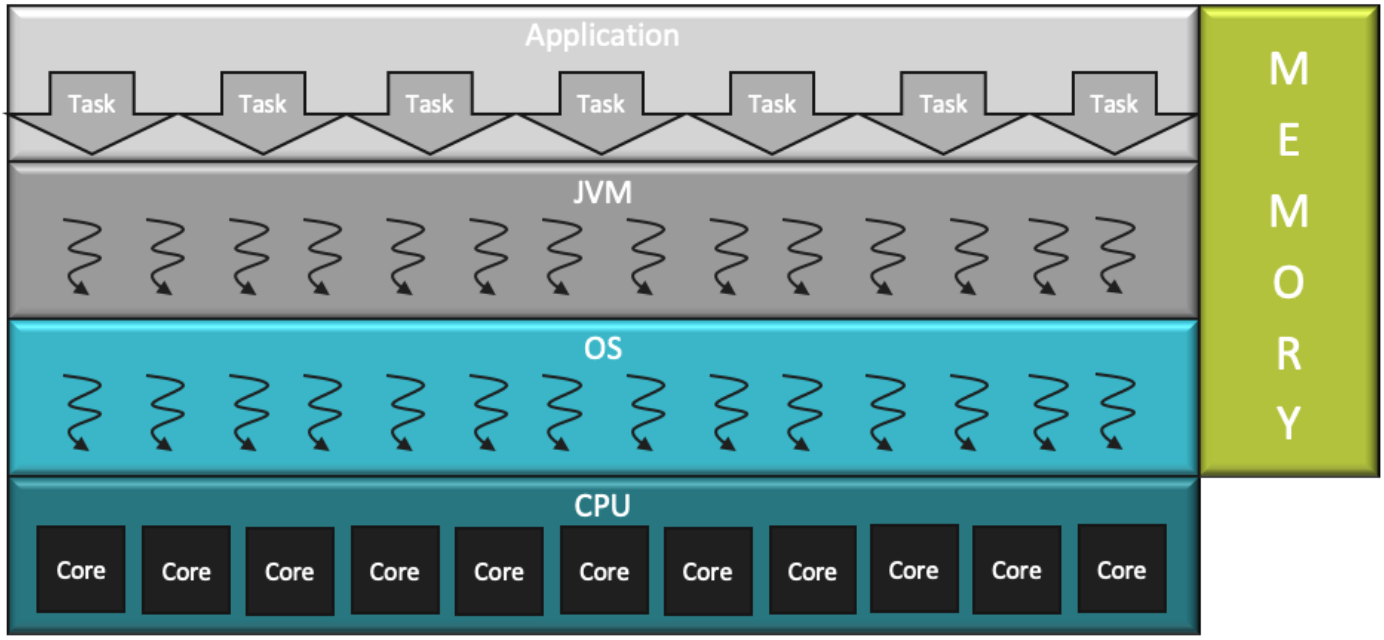
그럼 이 글에서 다루는 건 뭘까? 좋은 질문이다. 재미있는 건 위에서 다 빼버렸는데 무슨 얘기를 하려는 걸까?

똑같은 일을 하는 예제 하나를 옛날 방식부터 시작해서 자바 동시성 진화 과정을 따라 더 새롭고 색다른 방법으로 구현하면서 그 실행 '동작'(behavior)을 비교해보려고 한다. 여기서 다루는 방식이 전부는 아니며 다른 방식도 있다. 자바 언어에서 제공하는 API만으로 구현할 수 있는 방법만을 모아서 살펴보려고 했는데, 주류라고 할 수 있는 리액티브 방식을 외면할 수 없어서 같이 다루기로 한다.

자바 스레드

본격적으로 시작하기 전에 자바의 스레드에 대해 몇 가지만 짚고 넘어가자.

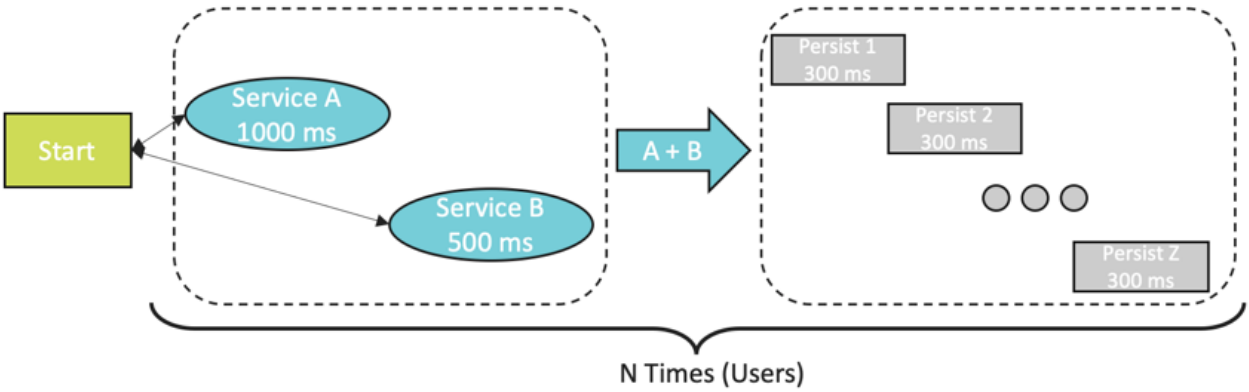
- JVM 스레드와 OS 스레드는 1:1로 매핑된다. JVM 스레드는 OS 스레드를 얇게 덧씌워 만든 거라고 볼 수 있다.
- OS는 아주 범용적인(그래서 느린) 스케줄링을 사용한다. OS는 JVM 내부에 대해 아무 것도 알지 못한다.
- 스레드를 만들고 이 스레드 저 스레드를 오가는 작업은 커널을 거쳐야 하므로 비용이 많이 든다(느리다).
- OS의 continuation 구현체는 자바 콜 스택(call stack) 뿐만 아니라 네이티브 콜 스택도 포함하며, 자원을 많이 사용한다.
- OS 스레드 갯수는 CPU 코어 숫자에 의해 제약받는다.
- 스레드에 사용되는 스택 메모리는 OS에 의해 힙 외의 영역에 마련된다.



태스크

예제에서 수행할 태스크(task)는 주로 동시에 호출되어 실행된다. 사용자별로 다음과 같은 흐름으로 요청을 처리하는 웹 서버를 떠올려보자.

- 서비스 A가 호출되면 완료될 때까지 1000ms가 필요하다.
- 서비스 B가 호출되면 완료될 때까지 500ms가 필요하다.
- 서비스 A, B의 결과는 파일, DB, S3 등 저장 방식별로 Z회 저장되며, 한 번 저장하는데 300ms가 필요하다. 현실에서는 저장 방식별로 소요 시간이 다르겠지만 계산의 편의를 위해 같다고 가정한다.



	No Concurrency	Full Concurrency
Execution Time	N * (SA + SB + Z*Persist)	SA + Persist
N=10, Z=3	24.000 ms	1.300 ms
N=1000, Z=30	10.500.000 ms (~3 hours)	~1.300 ms

동시 실행이 전혀 없는 No Concurrency 방식에서는 `요청 갯수 * (서비스A 처리 시간 + 서비스B 처리 시간 + 저장 횟수 * 저장 시간)`만큼의 시간이 필요하다.

반면에 모든 요청이 동시에 실행되는 이상적인 Full Concurrency 방식에서는 `Max(서비스A 처리 시간, 서비스B 처리 시간) + 저장 시간` 만큼, 그러니까 1,300ms가 필요하다.

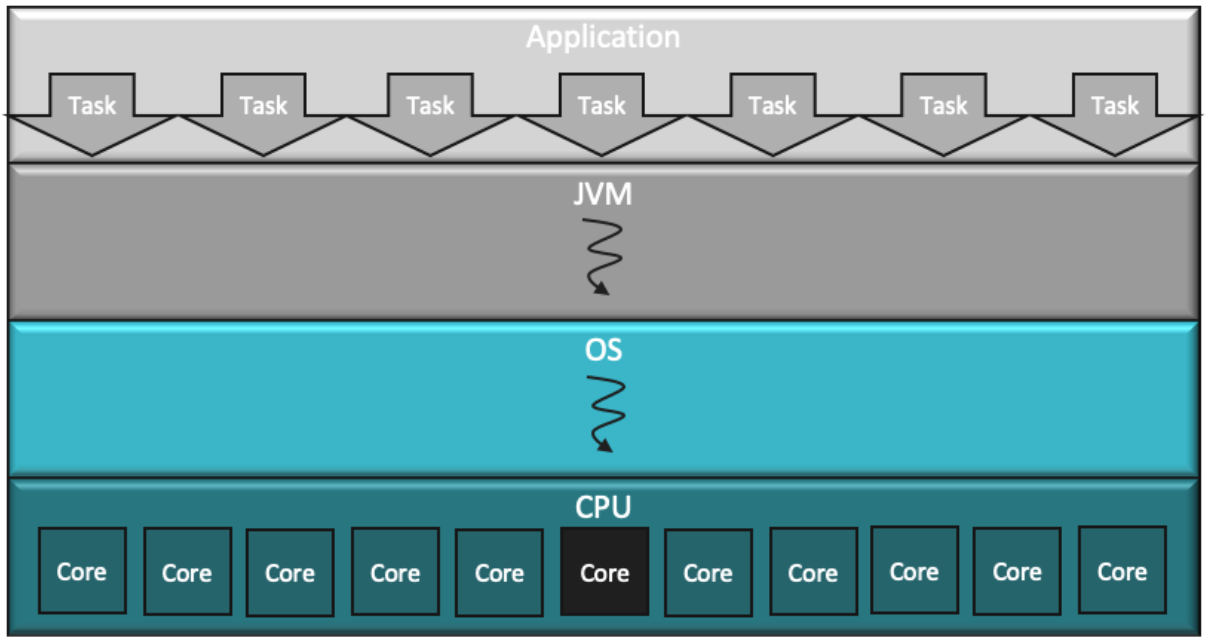
서비스 처리나 저장에 필요한 시간은 단순하게 `Thread.sleep()`으로 구현했다. 현실성은 물론 떨어지지만 여러 방식을 비교하고 이해하는 데는 충분하다.

전체 코드는 여기에서 확인할 수 있다. 요청 갯수(N)와 저장 횟수(Z)를 마음대로 바꿔가면서 어떤 결과가 나오는지 살펴보자. 늘 그렇듯이 모든 문제는 여러 가지 방식으로 풀 수 있으며, 깃헙에 있는 코드는 그 중 하나일 뿐이다. 가독성을 위해 최적화를 희생한 코드도 있음을 미리 밝혀둔다. 이제 예제 실행 결과를 살펴보면서 재미나는 자바 동시성 진화 과정을 따라가보자.

동시성 미사용

가장 단순한 방식이며, 아주 친숙한 코드다. 이런 코드 작성을 도와주는 도구도 많고, 쉬워서 직관적으로 금방 이해하고 디버깅 할 수 있다.

하지만 자원을 효율적으로 사용하지 못해서 실행 성능은 떨어진다는 것을 쉽게 알 수 있다. 사용된 유일한 JVM 스레드는 하나의 OS 스레드를 사용하며, 하나의 OS 스레드는 하나의 CPU 코어를 사용하므로 나머지 코어는 모두 놀게 된다. 요청 갯수와 저장 횟수가 많아지면 실행 소요 시간도 계속 늘어난다.



Code

코드도 아주 직관적이다. 짧고 핵심만 들어있다.

```
1  public void shouldBeNotConcurrent() {
2
3      for (int user = 1; user <= USERS; user++) {
4          String serviceA = serviceA(user);
5          String serviceB = serviceB(user);
6          for (int i = 1; i <= PERSISTENCE_FORK_FACTOR; i++) {
7              persistence(i, serviceA, serviceB);
8          }
9      }
10 }
```

helper 메서드 내용은 가독성을 위해 생략했는데 원한다면 여기에서 확인할 수 있다.

실행 결과

실행 결과는 앞의 'Task' 단원에 나온 그림 아래 쪽 표의 No Concurrency와 같다.

	No Concurrency
Execution Time	N * (SA + SB + Z*Persist)
N=10, Z=3	24.000 ms
N=1000, Z=30	10.500.000 ms (~3 hours)

네이티브 멀티 스레딩

멀티 스레딩에는 일반적으로 다음과 같은 난관이 있다.

- CPU 코어나 메모리 같은 자원의 효율적 이용
- 세밀한 스레드 갯수 조절이나 스레드 관리

- 제어 흐름과 컨텍스트 유실
- 실행 동기화
- 디버깅과 테스트

이 중 세 번째 항목인 '제어 흐름과 컨텍스트 유실'이 발생하는 이유는, 스택 트레이스가 요청 처리 전체를 아우르는 트랜잭션이 아니라, 그 중 일부만을 처리하는 스레드에 바운드 되므로, 현재 스레드에 할당된 일부 단계에 대한 정보만 스택 트레이스로 확인할 수 있기 때문이다. 그래서 디버깅이나 프로파일링이 어려워질 수 있다.

코드

무엇보다도 일단 코드 량이 확연히 늘어난 것을 볼 수 있다. `Runnable`을 구현해야 하고, 수동으로 스레드를 생성하고 제어하며, 동기화도 필요하다. 로직이 여기저기 흩어져 있어 따라가기가 어렵다.

반면에 실행 성능은 꽤 좋다. 스레드 생성과 컨텍스트 스위칭 비용이 있으므로 이상적인 수치인 1,300ms에는 못 미치지만 앞서 살펴본 '동시성 미사용' 방식에 비하면 훨씬 좋다.

아래는 가독성을 위해 일부만 가져왔으며 네이티브 멀티 스레딩 방식 전체 코드는 여기에서 확인할 수 있다.

```
1  public void shouldExecuteIterationsConcurrently() throws InterruptedException {
2
3      List<Thread> threads = new ArrayList<>();
4      for (int user = 1; user <= USERS; user++) {
5          Thread thread = new Thread(new UserFlow(user));
6          thread.start();
7          threads.add(thread);
8      }
9      // 종료 조건 - 가장 효율적인 방법은 아니지만 의도대로 동작한다.
10     for (Thread thread : threads) {
11         thread.join();
12     }
13 }
14
15 static class UserFlow implements Runnable {
16
17     private final int user;
18     private final List<String> serviceResult = new ArrayList<>();
19
20     UserFlow(int user) {
21         this.user = user;
22     }
23
24     @SneakyThrows
25     @Override
26     public void run() {
27         Thread threadA = new Thread(new Service(this, "A", SERVICE_A_LATENCY, user));
28         Thread threadB = new Thread(new Service(this, "B", SERVICE_B_LATENCY, user));
29         threadA.start();
30         threadB.start();
31         threadA.join();
32         threadB.join();
33
34         List<Thread> threads = new ArrayList<>();
35         for (int i = 1; i <= PERSISTENCE_FORK_FACTOR; i++) {
36             Thread thread = new Thread(new Persistence(i, serviceResult.get(0), serviceResult.get(1)));
37             thread.start();
38             threads.add(thread);
39         }
40
41         // 종료 조건 - 가장 효율적인 방법은 아니지만 의도대로 동작한다.
42         for (Thread thread : threads) {
43             thread.join();
44         }
45     }
46
47     public synchronized void addToResult(String result) {
48         serviceResult.add(result);
49     }
50 }
51 // Service와 Persistence 구현 코드는 생략
```

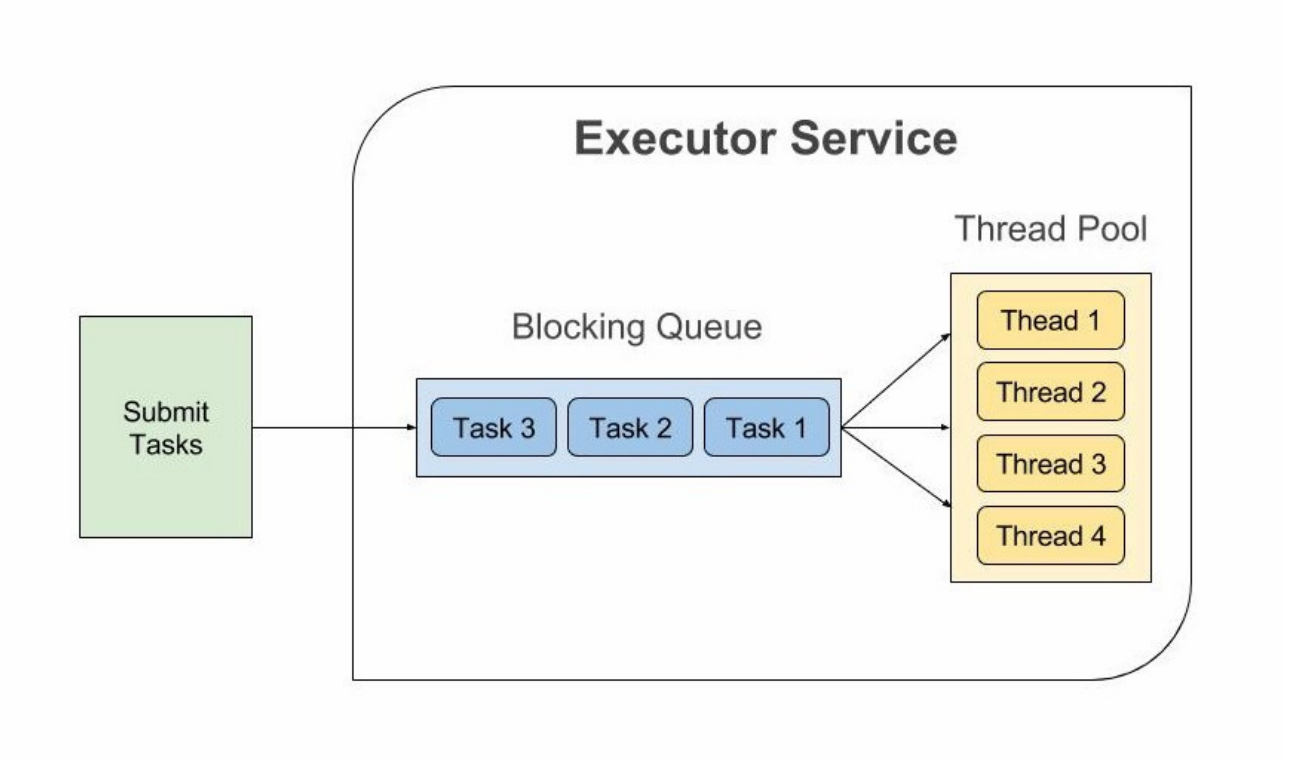
재미있는 사실

스레드를 무한정 생성할 수는 없다. OS마다 다르며 필자의 64비트 시스템에서 스레드 하나 당 1MB의 메모리(스레드 스택에 사용되는 메모리)를 점유한다. 요청 1000개, 30회 저장으로 설정하고 실행하면 33,000개의 스레드를 생성하려다가 Out Of Memory 에러가 발생한다.

Iteration / Persistence Fork Factor	Threads	Total Time (target 1300ms)
1000 / 3	6000	~4800ms
1000 / 30	33000	Out of Memory

ExecutorService

자바 5에 `ExecutorService`가 도입됐다. 스레드 풀링을 통해 새 스레드 생성 부담을 덜고 스레드를 로우 레벨로 다루는 부담을 덜어내는 게 주된 목표였다. 태스크는 `ExecutorService`에 submit 되고, 큐에 들어간다. 작업 가능한 스레드가 큐에서 태스크를 가져가서 실행한다.



눈여겨 볼 점은 다음과 같다.

- JVM 스레드 갯수가 여전히 OS 스레드 갯수에 의해 제한을 받는다.
- 스레드 풀에서 스레드를 하나 가져가면, 그 스레드는 연산을 수행하지 않더라도 다른 곳에 사용되지 못하고 낭비된다.
- `Future`가 반환되므로 발전된 것 같아 보이지만 조립(compose)할 수 없으며, 반환값을 얻기 위해 `get()`을 호출하면 태스크가 완료될 때까지 블로킹 된다.

코드

대체로 앞에서 살펴본 '네이티브 멀티 스레딩'과 비슷하다. 가장 큰 차이점은 스레드를 직접 생성하지 않고, 태스크를 `ExecutorService`에 submit 한다는 점이다. 스레드 풀 생성과 관리는 `ExecutorService`가 담당한다.

또 다른 점은 서비스 A와 서비스 B를 동기화하기 위해 `join()`을 사용하지 않는다는 점이다. 대신에 반환되는 `Future`의 `get()`을 호출해서 값이 반환될 때까지 블로킹한다.

예제에서는 2,000개의 스레드를 가진 스레드 풀이 사용됐다.

```
1  public void shouldExecuteIterationsConcurrently() throws InterruptedException {
2
3      for (int user = 1; user <= USERS; user++) {
4          executor.execute(new UserFlow(user));
5      }
6
7      // 종료 조건
8      latch.await();
9      executor.shutdown();
10     executor.awaitTermination(60, TimeUnit.SECONDS);
11 }
```

```
12
13 static class UserFlow implements Runnable {
14
15     private final int user;
16
17     UserFlow(int user) {
18         this.user = user;
19     }
20
21     @SneakyThrows
22     @Override
23     public void run() {
24         Future<String> serviceA = executor.submit(new Service("A", SERVICE_A_LATENCY, user));
25         Future<String> serviceB = executor.submit(new Service("B", SERVICE_B_LATENCY, user));
26
27         for (int i = 1; i <= PERSISTENCE_FORK_FACTOR; i++) {
28             executor.execute(new Persistence(i, serviceA.get(), serviceB.get()));
29         }
30
31         latch.countDown();
32     }
33 }
34 // Service와 Persistence 구현 코드는 생략
```

재미있는 사실

2천 개의 스레드를 사용했을 때 앞서 살펴본 '네이티브 멀티 스레딩'에 비해 훨씬 나은 성능을 보이고 있다.

Iteration / Persistence Fork Factor	Threads	Total Time (target 1300ms)
1000 / 3	2000	~2383ms
1000 / 30	2000	~4968ms

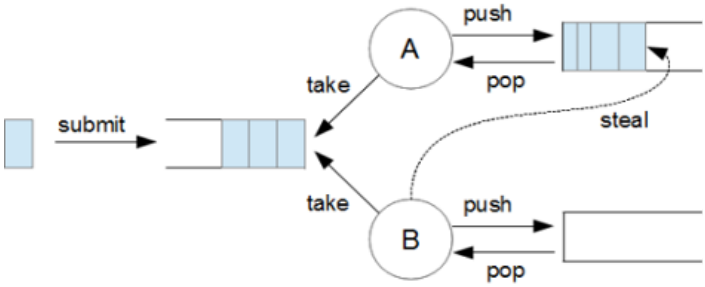
스레드 풀 크기를 제대로 설정하지 않으면 데드락이 발생하기도 한다. 풀 크기를 예를 들어 10정도로 작게 잡고 요청 갯수와 저장 횟수를 늘리면 데드락 때문에 아무 것도 제대로 완료되지 않는 것을 볼 수 있다. 왜냐하면 하나의 `UserFlow`가 다수의 스레드를 필요로 하도록 구현돼 있고, 다수의 `UserFlow`를 `ExecutorService`에 submit 하므로 풀에 있는 모든 스레드를 점유하게 되어 태스크를 완료할 수 있는 스레드가 남아있지 않기 때문이다.

예제에서는 스레드 풀 크기가 고정된 `fixedThreadPool`을 사용했지만, 크기가 동적으로 변하는 다른 스레드 풀을 사용할 수도 있다.

Fork/Join 프레임워크

자바 7에서 `ExecutorService` 기반으로 만들어진 Fork/Join 프레임워크가 도입됐다. Fork/Join 프레임워크는 재귀적으로 더 작은 크기로 쪼갤 수 있는 태스크를 효율적으로 처리하기 위해 만들어졌다. Fork/Join 프레임워크가 `ExecutorService`를 대체할 거라는 기대도 있었지만, Fork/Join 프레임워크는 동시 실행에 대해 개발자가 제어할 수 있는 옵션이 더 적으므로 `ExecutorService`는 여전히 계속 사용되고 있다.

`ExecutorService`와 확연히 다른 점은 작업 빼가기(work-stealing)다.



스레드 풀에 있던 스레드 A에 과부하가 걸려서 A 스레드 내부 큐가 꽉 차 있을 때, 스레드 풀에 있는 다른 스레드 B가 `ExecutorService`의 메인 큐에 있는 태스크를 가져오는 대신에 과부하 걸린 스레드 A 내부 큐에 있는 태스크를 가져와서 처리할 수 있다.

코드

스트림과 람다식 덕분이기도 하지만, 코드가 전체적으로 점점 짧아지고 있다.

Fork/Join 풀에 태스크를 submit 할 수 있는 `UserFlowRecursiveAction`를 구현했다. 예제 태스크가 재귀적으로 분할될 성질이 아니기 때문에 이런 태스크에 Fork/Join 프레임워크를 사용하는 것은 사실 적합하지는 않다. 다만 앞서 다뤄온 예제들의 연장선상에서

Fork/Join 프레임워크가 어떻게 동작하는지 알아보자는 목적에는 부합한다.

```
1  public void shouldExecuteIterationsConcurrently() throws InterruptedException {
2
3      commonPool.submit(new UserFlowRecursiveAction(IntStream.rangeClosed(1, USERS)
4              .boxed()
5              .collect(Collectors.toList())));
6
7      // Stop Condition
8      commonPool.shutdown();
9      commonPool.awaitTermination(60, TimeUnit.SECONDS);
10 }
11
12 public static class UserFlowRecursiveAction extends RecursiveAction {
13
14     private final List<Integer> workload;
15
16     public UserFlowRecursiveAction(List<Integer> workload) {
17         this.workload = workload;
18     }
19
20     @Override
21     protected void compute() {
22
23         if (workload.size() > 1) {
24             commonPool.submit(new UserFlowRecursiveAction(workload.subList(1, workload.size())));
25         }
26
27         int user = workload.get(0);
28
29         ForkJoinTask<String> taskA = commonPool.submit(() -> service("A", SERVICE_A_LATENCY, user));
30         ForkJoinTask<String> taskB = commonPool.submit(() -> service("B", SERVICE_B_LATENCY, user));
31
32         IntStream.rangeClosed(1, PERSISTENCE_FORK_FACTOR)
33                 .forEach(i -> commonPool.submit(() -> persistence(i, taskA.join(), taskB.join())));
34     }
35 }
```

재미있는 사실

앞서 'ExecutorService' 단원에서 해봤던 것처럼 이번에도 풀 크기를 10 정도로 잡게 잡고 실행해보자. 이번에는 Fork/Join 프레임워크의 작업 빼가기 기능 덕분에 데드락이 발생하지 완료된다. 하지만 Fork/Join 프레임워크를 사용한다고해서 데드락이 항상 발생하지 않는 것은 아니다. Fork/Join 프레임워크에서는 태스크가 어떤 방식으로 더 작은 태스크로 분할될 수 있는지에 따라 데드락 발생 여부가 정해진다.

예제 태스크는 재귀적 분할에 딱 들어맞는 태스크가 아니며, Fork/Join 프레임워크에 추가된 로직 때문에 실행 성능은 좋지 않아서 `ExecutorService` 방식 보다도 오히려 떨어진다. 하지만 작은 풀 사이즈에서도 데드락이 발생하지 않았으므로 안정성은 더 높다.

Iteration / Persistence Fork Factor	Threads	Total Time (target 1300ms)
1000 / 3	2000	~3581ms
1000 / 30	2000	~26124ms

CompletableFuture

자바 8에서 도입된 `CompletableFuture`는 Fork/Join 프레임워크를 기반으로 만들어졌다. 연산 결과를 모아서(combine) 처리할 수 있는 메서드가 하나도 없었고, 에러 처리를 위한 방법도 없었던 `Future` 인터페이스 도입 이후로 오랫동안 기다려 왔던 진화가 `CompletableFuture`에서 드디어 이루어졌다.

`CompletableFuture`를 통해 개선된 점은 다음과 같다.

- 더 개선된 함수형 프로그래밍 스타일 도입
- 로직을 조립(compose)하고, 결과를 모아서 처리(combine)하고, 비동기 연산 과정을 실행하고, 에러를 처리할 수 있는 50여개의 메서드 추가
- `CompletableFuture`의 평문형 API(fluent API) 대부분은 뒤에 `Async` 접미사가 붙은 것과 붙지 않은 것, 이렇게 2가지씩 짝지어져 있다. `Async` 접미사가 붙은 메서드는 해당 연산을 다른 스레드에서 실행하려고 할 때 사용된다.

코드

앞에서 다룬 'Fork/Join 프레임워크'보다도 훨씬 더 짧아졌고 압축적이다.

분량뿐 아니라 코딩 스타일 자체에서도 주목할만한 패러다임 변화가 눈에 띈다. 비동기 실행 결과를 모아서 처리하는 작업이 훨씬 자연스러운 코드로 표현된다.

하지만 함수형 프로그래밍 스타일에 익숙하지 않은 사람들에게는 대단히 생소해 보일 수도 있으며, 금방 적응하기 어려울 수도 있다.

```
public void shouldExecuteIterationsConcurrently() throws InterruptedException, ExecutionException {
    CompletableFuture.allOf(IntStream.rangeClosed(1, USERS)
            .boxed()
            .map(this::userFlow)
            .toArray(CompletableFuture[]::new)
    ).get();
}

@SneakyThrows
private CompletableFuture<String> userFlow(int user) {
    return CompletableFuture.supplyAsync(() -> serviceA(user), commonPool)

            .thenCombine(CompletableFuture.supplyAsync(() -> serviceB(user), commonPool), this::persist);
}

@SneakyThrows
private String persist(String serviceA, String serviceB) {
    CompletableFuture.allOf(IntStream.rangeClosed(1, PERSISTENCE_FORK_FACTOR)
            .boxed()
            .map(iteration -> CompletableFuture.runAsync(() -> persistence(iteration, serviceA, serviceB), commonPool))
            .toArray(CompletableFuture[]::new)
    ).join();

    return "";
}
```

재미있는 사실

`CompletableFuture`가 Fork/Join 프레임워크를 바탕으로 만들어졌음에도 불구하고 실행 성능은 훨씬 좋다.

Iteration / Persistence Fork Factor	Threads	Total Time (target 1300ms)
1000 / 3	2000	~2170ms
1000 / 30	2000	~8600ms

`CompletableFuture`는 자바에 있는 몇 안 되는 모나드(monad) 중 하나다. 모나드를 알아보려면 이상한 나라의 앨리스에 나오는 토끼굴에 들어가는 모험을 감수해야 하므로 이 글에서는 다루지 않는다.

Reactive

리액티브 얘기를 이어가지 전에 먼저 반드시 구분해둬야 할 것이 있다. 리액티브 아키텍처와 리액티브 프로그래밍은 완전히 다르다는 점이다. 이 글에서는 비동기 데이터 스트림을 처리하고 에러 처리와 배압(backpressure)를 확고하게 지원하는 리액티브 프로그래밍만을 다룬다.

`CompletableFuture`가 진화해서 리액티브 방식이 됐다고 볼 수도 있지만 사실은 물론 그 이상이다. 리액티브 프로그래밍의 주요 목표는 프로그램 구조를 비동기 이벤트 스트림으로 재구성하는 것이며, 스레드 관리는 라이브러리/프레임워크에 위임한다.

주목할 점은 다음과 같다.

- 데이터를 발생시키는 `Observable`, 데이터를 소비하는 `Observer`, 스레드를 관리하는 `Scheduler`의 삼위 일체
- 리액티브 방식을 도입하면 프로그램 흐름 전부가 리액티브 방식으로 같이 바뀌어야 한다는 점에서 전염성이 강하다. 일부에 블로킹 코드가 남아 있으면 리액티브의 장점은 전혀 발휘되지 못한다.
- 리액티브 구현체도 여러가지가 있다. 처음에는 RxJava가 있었지만 최근에는 스프링의 Reactor가 대세다. 액터 모델을 구현하는 Akka 프레임워크는 RxJava나 Reactor보다 더 급진적인 리액티브 프로그래밍을 적용하고 있다.

코드

리액티브는 이 글에서는 유일하게 자바 언어 자체적으로는 제공되지 않는 솔루션이다. 예제에서는 스프링 Reactor를 사용했지만 RxJava도 크게 다르지 않다.

명령형 코딩 스타일에만 익숙하다면 이제 리액티브 코드가 아주 생소해 보일 수 있다. 리액티브 코드에 익숙해지려면 단순히 코딩뿐 아니라 테스트와 디버깅에서도 마인드셋을 바꿔야 한다. 특히 리액티브 코드의 테스트와 디버깅은 상당히 어렵지만 충분히 투자해볼 만 하다.

리액티브 코딩 스타일을 마스터하면 지속적으로 발전하는 리액티브 지원 라이브러리의 도움에 힘입어 대단히 성능이 좋은 코드를 효율적으로 작성할 수 있다.

```
1  public void shouldExecuteIterationsConcurrently() {
2      Flux.range(1, USERS)
3              .flatMap(i -> Mono.defer(() -> userFlow(i)).subscribeOn(Schedulers.parallel()))
4              .blockLast();
5  }
6
7  private Mono<String> userFlow(int user) {
8
9      Mono<String> serviceA = Mono.defer(() -> Mono.just(serviceA(user))).subscribeOn(Schedulers.elastic());
10     Mono<String> serviceB = Mono.defer(() -> Mono.just(serviceB(user))).subscribeOn(Schedulers.elastic());
11
12     return serviceA.zipWith(serviceB, (sA, sB) -> Flux.range(1, PERSISTENCE_FORK_FACTOR)
13             .flatMap(i ->
14                     Mono.defer(() -> Mono.just(persistence(i, sA, sB))).subscribeOn(Schedulers.elastic())
15             )
16             .blockLast()
17     );
18 }
```

재미있는 사실

작업 처리를 더 세밀하게 제어할 수 있게 됐지만, 스레드 풀 튜닝은 알아서 수행되므로 개발자가 직접 할 필요는 없다. 예제 코드에서는 리액티브의 진정한 장점 중 하나인 에러 처리나 배압 처리는 사용되지 않았다.

실행 성능은 `CompletableFuture`에 비해 떨어진다.

Iteration / Persistence Fork Factor	Threads	Total Time (target 1300ms)
1000 / 3	elastic	~5635ms
1000 / 30	elastic	~10070ms

Project Loom

프로젝트 룸(Loom)은 아직 정식 출시되지 않았다. 2020년 12월 현재 기준 자바 16 Early Access 버전에 포함돼서 이것저것 실제로 실험해볼 수는 있지만 자바 16에 포함될지는 미지수다.

프로젝트 룸은 가상 스레드(Virtual Thread)와 관련된 여러 기능이 포함돼 있다. 그 중에서 '가상 스레드'와 '구조적 동시성(Structured Concurrency)'만 이 글에서 다룬다.

가상 스레드를 사용하면 `JVM 스레드 : OS 스레드 = 1 : 1`라는 오랜 등식이 더이상 성립되지 않는다. 가상 스레드는 기존의 JVM 스레드에 비해 훨씬 가볍고 저렴하다. 구조적 동시성을 도입하면 스레드 라이프타임이 스레드가 사용된 코드 블록과 연관(correlate)돼서 동기화가 훨씬 분명해지고, 우리가 익숙한 명령형 코딩 스타일을 그대로 사용할 수 있다.

주목해볼 점은 다음과 같다.

- 메타데이터, 스택 메모리, 컨텍스트 스위치 시간이 네이티브 OS 스레드의 수 분의 일 밖에 되지 않을만큼 가볍다.
- 아직 지원 도구가 충분하지 않다. 프로젝트 룸을 사용할 때 IntelliJ, Gradle, Lombok, JProfiler 등에서 여러가지 이슈가 발생했다. 그래서 프로젝트 룸 예제 코드는 별도의 프로젝트로 따로 작성했다.

코드

동시성을 전혀 사용하지 않은 단순하고 쉬운 코드와는 차이가 좀 있지만, 동시성을 사용했던 다른 코드들보다는 훨씬 단순하고 친숙해보인다.

가상 스레드를 지원하는 새로운 `ExecutorService` 구현체가 있고, `AutoCloseable` 인터페이스를 구현하고 있어서 try-with-resource 구문으로 사용해서 안전하게 자원을 열고 닫을 수 있다.

구조적 동시성이 적용돼 있어서 부모 스레드는 자기가 생성한 모든 자식 스레드의 종료를 try 블록 안에서 기다린다.

```
1  @SneakyThrows
2  private void startConcurrency() {
3      try (var e = Executors.newVirtualThreadExecutor()) {
4          IntStream.rangeClosed(1, USERS).forEach(i -> e.submit(() -> userFlow(i)));
5      }
6  }
7
```

```
@SneakyThrows
private void userFlow(int user) {
    List<Future<String>> result;
    try (var e = Executors.newVirtualThreadExecutor()) {
        result = e.invokeAll(List.of(() -> serviceA(user),() -> serviceB(user)));
    }

    persist(result.get(0).get(), result.get(1).get());
}

private void persist(String serviceA, String serviceB) {
    try (var e = Executors.newVirtualThreadExecutor()) {
        IntStream.rangeClosed(1, PERSISTENCE_FORK_FACTOR)
                .forEach(i -> e.submit(() -> persistence(i, serviceA, serviceB)));
    }
}
```

재미있는 사실

JVM 차원에서의 개선이기 때문에 도입되면 많은 레거시 애플리케이션/라이브러리들이 별다른 수정 없이도 성능 개선 효과를 그대로 누릴 수 있다.

표에서 나타난 것처럼 실제로 몇 개의 OS 스레드를 사용했는지는 알 수 없지만, OS 스레드 갯수가 중요한 것은 아니다.

중요한 것은 프로젝트 룸을 사용했을 때 예제 실행 총 소요 시간이 이상적인 수치인 1,300ms에 가장 가깝다는 점이다. 요청 갯수나 저장 횟수를 늘리더라도 앞에서 살펴봤던 다른 방식들처럼 뚜렷한 성능 저하를 보이지도 않는다. 따라서 프로젝트 룸의 확장성이 상당히 좋다고 얘기할 수 있다.

Iteration / Persistence Fork Factor	Threads	Total Time (target 1300ms)
1000 / 3	???	~1448ms
1000 / 30	???	~1941ms

결론

동시성 처리가 복잡하다는 건 분명한 사실이다. 동시성을 적용하지 않았던 '동시성 미사용' 예제에 비해 '네이티브 멀티 스레딩' 예제 코드는 훨씬 복잡하고 읽기도 어려우며 디버그하기는 거의 불가능했다.

하지만 동시성 처리 전문가가 되고 싶지 않거나 되고 싶더라도 시간이 없었던 개발자들에게도 마침내 희망의 등불이 켜졌다. 프로젝트 룸 덕분에 동시성 처리 전문가가 되지 않더라도 충분히 성능 좋은 코드를 작성할 수 있게 됐다.

프로젝트 룸이 출시되면 리액티브 프로그래밍을 완전히 대체할지는 아직 알 수 없다. 만병통치약은 존재하지 않는다는 사실을 감안할 때 프로젝트 룸과 리액티브 프로그래밍은 공존할 가능성이 높다고 본다. 다양한 지원 도구를 갖고 있는 리액티브 프로그래밍으로 해결하는 것이 더 나은 문제도 있을 것이다. 아직까지는 지원 도구가 많지 않지만 자바 동시성 진화 과정에서 더 자연스럽고 더 친숙한 다음 단계는 프로젝트 룸이라고 할 수 있다.

예제 성능 측정치는 모두 필자의 로컬 장비에서 수행된 결과이므로 과학적인 검증을 거친 측정치라고 볼 수는 없다. 하지만 다양한 방식을 비교하는 목적으로는 의미있는 수치라고 할 수 있다.

프로젝트 룸 적용 코드는 여기에서, 룸을 사용하지 않은 다른 코드는 여기에서 확인할 수 있다.
