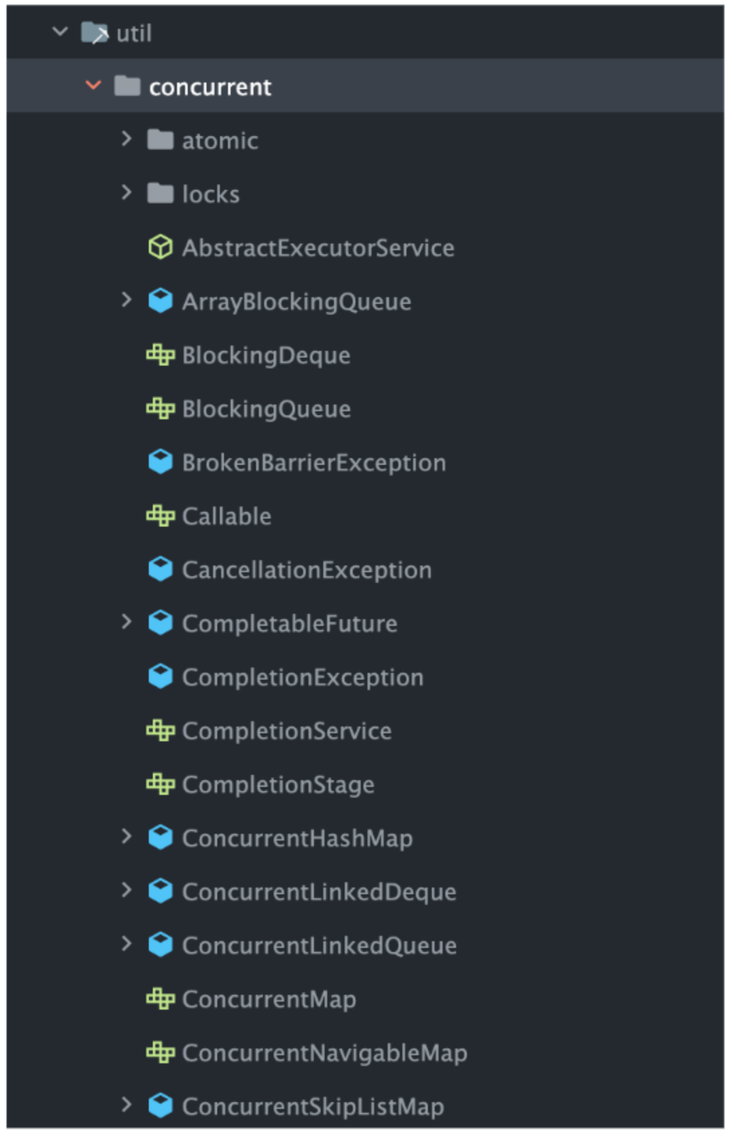
item 80

스레드보다는 실행자, 태스크, 스트림을 애용하자

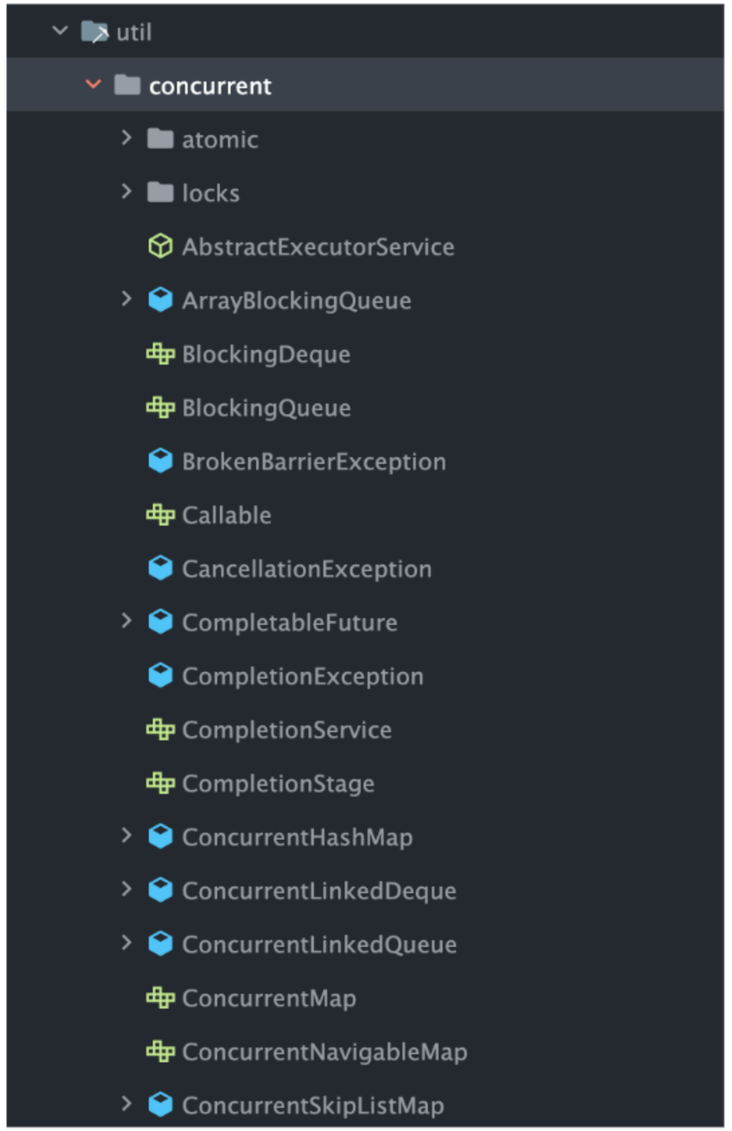
유현우

실행자 프레임워크

- Executor Framework는 Java 5에 도입된 java.util.concurrent 패키지의 일부



실행자 프레임워크



java.util.concurrent.Executors

```
public class Executors {

    Creates a thread pool that reuses a fixed number of threads operating off a shared unbounded
    queue. At any point, at most nThreads threads will be active processing tasks. If additional tasks
    are submitted when all threads are active, they will wait in the queue until a thread is available. If
    any thread terminates due to a failure during execution prior to shutdown, a new one will take its
    place if needed to execute subsequent tasks. The threads in the pool will exist until it is explicitly
    shutdown.
    Params: nThreads – the number of threads in the pool
    Returns: the newly created thread pool
    Throws: IllegalArgumentException – if nThreads <= 0

  public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads,
                                  keepAliveTime: 0L, TimeUnit.MILLISECONDS,
                                  new LinkedBlockingQueue<Runnable>());
  }
```

실행자 프레임워크

```
// 큐를 생성한다.
ExecutorService exec = Executors.newSingleThreadExecutor();

// 실행자에 실행할 task를 넘기는 방법 (실행)
exec.execute(runnable);

// 실행자 종료
exec.shutdown();
```

과거에는 단순한 작업 큐(work queue)를 만들기 위해서 수많은 코드를 작성해야 했는데,
간단하게 작업 큐를 생성할 수 있다.

실행자 프레임워크의 주요 기능

submit().get()

```
public static void main(String[] args) throws ExecutionException,
InterruptedException {
    ExecutorService exec = Executors.newSingleThreadExecutor();
    Future<String> submit = exec.submit(() -> "test");
    submit.get();
  }
```

특정 태스크가 완료되기를 기다릴 수 있다. 즉, 비동기를 동기처리 하고싶을 때 사용할 수 있다.

Future란?

java.util.concurrent.Future

```
public interface Future<V> {

    Attempts to cancel execution of this task. This method has no effect if the task is already
    completed or cancelled, or could not be cancelled for some other reason. Otherwise, if this task has
    not started when cancel is called, this task should never run. If the task has already started, then
    the mayInterruptIfRunning parameter determines whether the thread executing this task (when
    known by the implementation) is interrupted in an attempt to stop the task.

    The return value from this method does not necessarily indicate whether the task is now cancelled;
    use isCancelled.

    Params: mayInterruptIfRunning – true if the thread executing this task should be interrupted
            (if the thread is known to the implementation); otherwise, in-progress tasks are allowed
            to complete
    Returns: false if the task could not be cancelled, typically because it has already completed; true
             otherwise. If two or more threads cause a task to be cancelled, then at least one of them
             returns true. Implementations may provide stronger guarantees.

  boolean cancel(boolean mayInterruptIfRunning);

    Returns true if this task was cancelled before it completed normally.

    Returns: true if this task was cancelled before it completed

  boolean isCancelled();
```

Future의 주요 메소드

Future 인터페이스에는 다음과 같은 주요 메소드들이 있습니다.

get(): 연산의 결과를 반환합니다.
만약 연산이 아직 완료되지 않았다면, 완료될 때까지 기다립니다.
이 메소드는 Future가 가지는 제네릭 타입의 객체를 반환합니다.

get(long timeout, TimeUnit unit): 지정한 시간 동안만 결과를 기다리고, 그 시간이 지나면 TimeoutException을 던집니다. 이 메소드는 시간이 지나면 TimeoutException을 던지며, 그 전에 작업이 완료되면 그 결과를 반환합니다.

isDone(): 연산이 완료되었는지의 여부를 반환합니다.
작업이 완료되었다면 true, 그렇지 않다면 false를 반환합니다.

cancel(boolean mayInterruptIfRunning): 연산을 취소합니다.
매개 변수는 작업이 진행 중일 때 중단해도 되는지를 결정합니다.

isCancelled(): 작업이 취소되었는지의 여부를 반환합니다.
작업이 취소되었다면 true, 그렇지 않다면 false를 반환합니다.

Java에서 Future는 비동기 계산의 아직 계산되지 않은 결과를 표현하는 인터페이스입니다.

실행자 프레임워크의 주요 기능

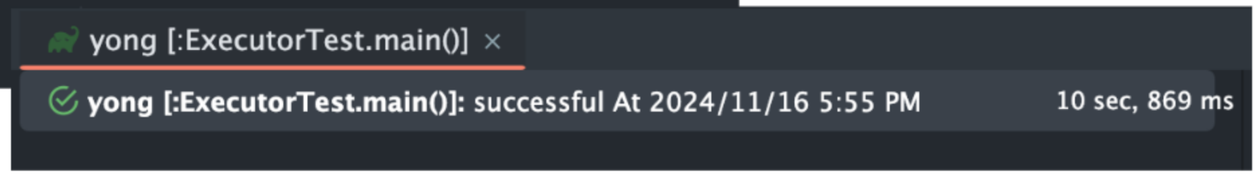
```java
public static void main(String[] args) throws ExecutionException, InterruptedException {
    ExecutorService exec = Executors.newSingleThreadExecutor();
    List<Callable<String>> tasks = List.of(() -> "test1", () -> "test2");

    List<Future<String>> futures = exec.invokeAll(tasks);
    System.out.println("All Tasks done");
    futures.forEach(f -> System.out.println(f.isDone()));

    System.out.println(exec.invokeAny(tasks));;
    System.out.println("Any Task done");

    exec.shutdown();
  }
```

```
> Task :ExecutorTest.main()
All Tasks done
true
true
test1
Any Task done
```

태스크 모음 중에서 어느 하나(invokeAny) 혹은 모든 태스크(invokeAll)가 완료되는 것을 기다릴 수 있다.

실행자 프레임워크의 주요 기능

```java
public static void main(String[] args) throws ExecutionException, InterruptedException {
    ExecutorService exec = Executors.newSingleThreadExecutor();
    Future<String> future = exec.submit(() -> "test");
    exec.awaitTermination( timeout: 10, TimeUnit.SECONDS);
    exec.shutdown();
}
```



Blocks until all tasks have completed execution after a shutdown request, or the timeout occurs, or the current thread is interrupted, whichever happens first.

Params: `timeout` - the maximum time to wait
`unit` - the time unit of the timeout argument

Returns: `true` if this executor terminated and `false` if the timeout elapsed before termination

Throws: `InterruptedException` - if interrupted while waiting

```java
boolean awaitTermination(long timeout, @NotNull TimeUnit unit)
    throws InterruptedException;
```

실행자 서비스가 종료하기를 기다리는데, 정확히는 지정된 시간 동안 스레드가 차단됩니다.

실행자 프레임워크의 주요 기능

```java
public static void main(String[] args) throws ExecutionException, InterruptedException {
    final int MAX_SIZE = 3;
    ExecutorService executorService = Executors.newFixedThreadPool(MAX_SIZE);
    ExecutorCompletionService<String> executorCompletionService =
        new ExecutorCompletionService<>(executorService);

    List<Future<String>> futures = List.of(
        executorCompletionService.submit(() -> "test1"),
        executorCompletionService.submit(() -> "test2"),
        executorCompletionService.submit(() -> "test3")
    );

    for (int loopCount = 0; loopCount < MAX_SIZE; loopCount++) {
      try {
        String result = executorCompletionService.take().get();
        System.out.println(result);
      } catch (InterruptedException | ExecutionException e) {
        //
      }
    }
    executorService.shutdown();
  }
```

```
> Task :ExecutorTest.main()
test2
test1
test3
```

완료된 태스크들의 결과를 차례로 받는다.

실행자 프레임워크의 주요 기능

```java
public static void main(String[] args) throws ExecutionException, InterruptedException {
    ScheduledThreadPoolExecutor executor = new ScheduledThreadPoolExecutor(1);

    executor.scheduleAtFixedRate(() -> {
      System.out.println(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss")
          .format(LocalDateTime.now()));
    }, 0, 2, TimeUnit.SECONDS);

    executor.awaitTermination(6, TimeUnit.SECONDS);
    executor.shutdown();
  }
```

```
> Task :ExecutorTest.main()
2024-11-16 18:10:06
2024-11-16 18:10:08
2024-11-16 18:10:10
2024-11-16 18:10:12
```

태스크를 특정 시간에 혹은 주기적으로 실행하게 한다.

ThreadPool 종류

```
public static void main(String[] args) throws ExecutionException, InterruptedException {
  ScheduledExecutorService scheduledExecutorService = Executors.newSingleThreadScheduledExecutor();
  ExecutorService executorService = Executors.newFixedThreadPool( nThreads: 10);
  ExecutorService executorService1 = Executors.newCachedThreadPool();
  ExecutorService executorService2 = Executors.newWorkStealingPool();
}
```

필요한 실행자 대부분은 java.util.concurrent.Executors의 정적 팩터리를 이용해 생성할 수 있다.

ThreadPool 종류

```
public static void main(String[] args) throws ExecutionException, InterruptedException {
    // 가벼운 프로그램을 실행하는 서버
    ExecutorService cachedThreadPool = Executors.newCachedThreadPool();
    // 무거운 프로그램을 실행하는 서버
    ExecutorService fixedThreadPool = Executors.newFixedThreadPool( nThreads: 10);
}
```

- Executors.newCachedThreadPool 요청받은 태스크를 큐에 쌓지 않고 바로 처리하며 사용 가능한 스레드가 없다면 즉시 스레드를 새로 생성해서 처리한다.
- 서버가 무겁다면 CPU 이용률이 100%로 치닫고 새로운 태스크가 도착할 때마다 다른 스레드를 생성하며 상황이 더 악화될 것이다.

스레드를 직접 다루지 말자

작업 큐를 직접 만들거나 스레드를 직접 다루는 것도 일반적으로 삼가야 한다.

스레드를 직접 다루지 말자

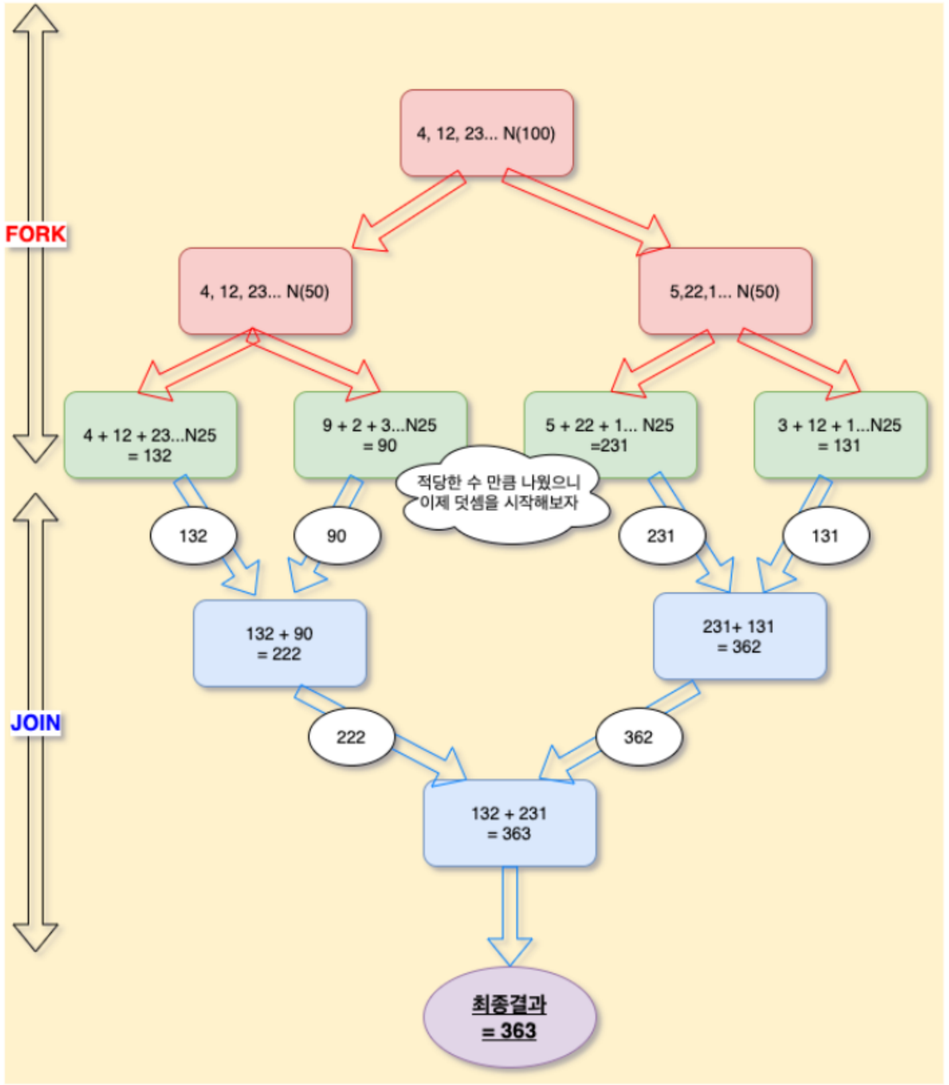
실행자 프레임워크를 이용하면 작업 단위와 실행 매커니즘을 분리할 수 있다.

```java
// () -> void
@FunctionalInterface
public interface Runnable {
    public abstract void run();
}

// () -> V
@FunctionalInterface
public interface Callable<V> {
    V call() throws Exception;
}
```

- 작업 단위는 Runnable과 Callable로 나눌 수 있다.
- Callable은 Runnable과 비슷하지만 값을 반환하고 임의의 예외를 던질 수 있다.

마지막으로

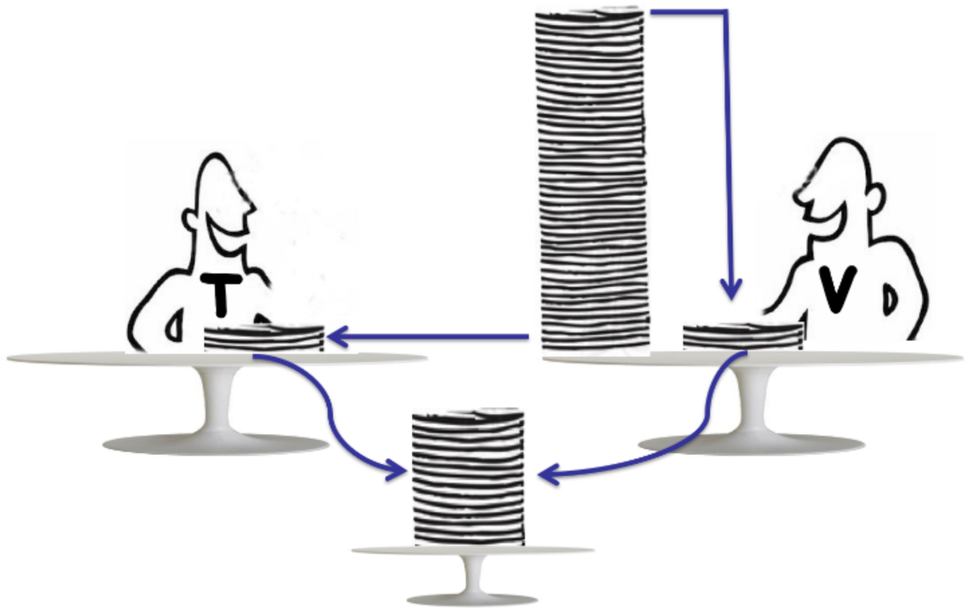
- 자바 7부터 실행자 프레임워크는 포크-조인(fork-join) 태스크를 지원하도록 확장됐다.



- ForkJoinTask의 인스턴스는 작은 하위 태스크로 나뉠 수 있다.
- ForkJoinPool을 구성하는 스레드들이 이 태스크들을 처리한다.
- 일을 먼저 끝낸 스레드가 다른 스레드의 남은 태스크를 가져와 대신 처리할 수도 있다.

마지막으로

- Fork/Join에서는 Work Stealing이라는 개념이 포함되어있다.



- Work Stealing은 양쪽 끝에서 넣고 뺄 수 있는 구조인 Dequeue와 관련이 있는 개념이다.
- 여러 개의 Dequeue에서 일이 진행될 때, 여유로운 Dequeue에서 바쁜 Dequeue의 일을 가져가서 해준다.

- 최대한의 CPU 활용을 뽑아내어 높은 처리량과 낮은 지연시간을 달성한다.
- 병렬 스트림도 이러한 ForkJoinPool을 이용하여 구현되어 있다.