Java Thread 를 관리하는 방법 -2-

CompletableFuture

> ⚠ Future 의 단점 및 한계
>
> 외부에서 완료시킬 수 없고, get 의 타임아웃 설정으로만 완료 가능
> get 과 같은 블로킹 코드를 통해서만 이후의 결과를 처리할 수 있음
> 여러 Future 를 조합할 수 없음
> 여러 작업을 조회하거나 예외 처리를 할 수 없음

위와 같은 Future 의 한계를 극복하기 위해 CompletableFuture 가 등장하게 됐다.

CompletableFuture 는 기존의 Future 를 기반으로 외부에서 완료시킬 수 있다.
CompletableFuture 는 Future 이외에도 CompletionStage 인터페이스도 구현하고 있는데, CompletionStage 는 작업들을 중첩시키거나 완료 후 콜백을 위해 추가되었다.

Future 에서는 불가능했던 '몇 초 이내에 응답이 안 오면 기본 값을 반환한다' 와 같은 작업이 가능해졌다. 즉, Future 의 진화된 형태로써 외부에서 작업을 완료시킬 수 있을 뿐 아니라 콜백 등록 및 Future 조합이 가능하다.

비동기 작업 실행

- `runAsync()`
  - 반환값이 없는 경우
  - 비동기로 작업 실행 콜
- `supplyAsync()`
  - 반환값이 있는 경우
  - 비동기로 작업 실행 콜

```
@Test
void runAsyncTest() throws ExecutionException, InterruptedException {
        CompletableFuture<Void> future = CompletableFuture.runAsync(() -> {
                System.out.println("Thread : " +
Thread.currentThread().getName());
        });

        future.get();
        System.out.println("Thread : " + Thread.currentThread().getName());
}

@Test
void supplyAsyncTest() throws ExecutionException, InterruptedException {
        CompletableFuture<String> futureWithReturn =
CompletableFuture.supplyAsync(() -> {
```

```
                return "Thread : " + Thread.currentThread().getName();
        });

        System.out.println(futureWithReturn.get());
        System.out.println("Thread : " + Thread.currentThread().getName());
}
```

`runAsync()` 와 `supplyAsync()` 는 기본적으로 Java 7 에 추가된 ForkJoinPool 의 `commonPool()` 을 사용해 작업을 실행할 스레드를 스레드 풀로부터 얻어 실행시킨다.

만약 원하는 스레드 풀을 사용하고자 한다면, ExecutorService 를 파라미터로 넘겨주면 된다.

```
@Test
void supplyAsyncWithAnotherPoolTest() throws ExecutionException,
InterruptedException {
        ExecutorService executorService = Executors.newSingleThreadExecutor();
        CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
                return "Thread : " + Thread.currentThread().getName();
        }, executorService);

        System.out.println(future.get());
        System.out.println("Thread : " + Thread.currentThread().getName());

        executorService.shutdown();

}
```

작업 콜백

- `thenApply()`
  - 반환 값을 받아서 다른 값을 반환한다
  - 함수형 인터페이스 `Function` 을 파라미터로 받는다
- `thenAccept()`
  - 반환 값을 받아 처리하고 값을 반환하지 않음
  - 함수형 인터페이스 `Consumer` 를 파라미터로 받는다
- `thenRun()`
  - 반환 값을 받지 않고 다른 작업을 실행한다
  - 함수형 인터페이스 `Runnable` 을 파라미터로 받는다

```
@Test
void thenApplyTest() throws ExecutionException, InterruptedException {
        CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
                return "Thread : " + Thread.currentThread().getName();
        }).thenApply(result -> result + " This is added By thenApply Function");


        System.out.println(future.get());
```

```
        System.out.println(future.get());
        System.out.println("Thread : " + Thread.currentThread().getName());
}

@Test
void thenAcceptTest() {
        CompletableFuture<Void> future = CompletableFuture.supplyAsync(() -> {
                return "Thread : " + Thread.currentThread().getName();
        }).thenAccept(result -> System.out.println(result + " This is added By
thenAccept Function"));

        System.out.println("Thread : " + Thread.currentThread().getName());
}

@Test
void thenRunTest() {
        CompletableFuture<Void> future = CompletableFuture.supplyAsync(() -> {
                return "Thread : " + Thread.currentThread().getName();
        }).thenRun(() -> System.out.println("This is added By thenRun
Function"));

        System.out.println("Thread : " + Thread.currentThread().getName());
}
```

작업 조합

- `thenCompose()`
  - 두 작업이 이어서 실행하도록 조합하며, 앞선 작업의 결과를 받아서 사용할 수 있다
  - 함수형 인터페이스 `Function` 을 파라미터로 받는다
- `thenCombine()`
  - 두 작업을 독립적으로 수행하고, 둘 모두 완료됐을 때 콜백을 실행한다
  - 함수형 인터페이스 `Function` 을 파라미터로 받는다
- `allOf()`
  - 여러 작업들을 동시에 수행하고 모든 작업 결과에 대한 콜백을 수행한다
- `anyOf()`
  - 여러 작업들 중 가장 빨리 수행된 하나의 결과에 콜백을 수행한다

```
@Test
void thenComposeTest() throws ExecutionException, InterruptedException {
        CompletableFuture<String> helloFuture = CompletableFuture.supplyAsync(()
-> {
                return "Hello ";
        });

        CompletableFuture<String> future =
helloFuture.thenCompose(this::introduceFuture);
        System.out.println(future.get());
```

```
}

private CompletableFuture<String> introduceFuture(String message) {
        return CompletableFuture.supplyAsync(() -> {
                return message + "I am Java";
        });
}

@Test
void thenCombineTest() throws ExecutionException, InterruptedException {
        CompletableFuture<String> helloFuture = CompletableFuture.supplyAsync(()
-> {
                return "Hello";
        });

        CompletableFuture<String> introduceFuture =
CompletableFuture.supplyAsync(() -> {
                return "I am Java";
        });

        CompletableFuture<String> future =
helloFuture.thenCombine(introduceFuture, (firstFuture, secondFuture) ->
                firstFuture + " " + secondFuture
        );
        System.out.println(future.get());
}
```

`thenCompose()` 는 `helloFuture()` 가 먼저 실행된 이후 반환 값을 매개변수로 하여 `introductFuture()` 가 실행된다.

`thenCombine()` 은 `helloFuture()` 와 `introduceFuture()` 각각 수행되고 얻은 반환 값들을 조합하여 작업을 처리한다.

```
@Test
void allOfTest() throws ExecutionException, InterruptedException {
        CompletableFuture<String> helloFuture = CompletableFuture.supplyAsync(()
-> {
                return "Hello";
        });

        CompletableFuture<String> introduceFuture =
CompletableFuture.supplyAsync(() -> {
                return "I am Java";
        });

        List<CompletableFuture<String>> futures = List.of(helloFuture,
introduceFuture);
```

```
        CompletableFuture<List<String>> resultFutures = CompletableFuture.allOf(
                                        futures.toArray(new
CompletableFuture[futures.size()]))
                                        .thenApply(result -> futures.stream()
                                                .map(CompletableFuture::join)
                                                .collect(Collectors.toList())
                                        );

        resultFutures.get().forEach(System.out::println);
}

@Test
void anyOfTest() throws ExecutionException, InterruptedException {
        CompletableFuture<String> helloFuture = CompletableFuture.supplyAsync(()
-> {
                try {
                        Thread.sleep(3000L);
                } catch (InterruptedException e) {
                        throw new RuntimeException(e);
                }

                return "Hello";
        });

        CompletableFuture<String> introduceFuture =
CompletableFuture.supplyAsync(() -> {
                return "I am Java";
        });

        CompletableFuture<Void> future = CompletableFuture.anyOf(helloFuture,
introduceFuture)
        .thenAccept(System.out::println);

        future.get();
}
```

`allOf()` 가 모든 결과에 대해 콜백이 적용되는 반면 `anyOf()` 는 가장 먼저 실행이 끝난 `introduceFuture()` 에만 콜백이 적용되는 것을 확인할 수 있다.

예외 처리

- `exceptionally()`
  - 발생한 에러를 받아서 예외를 처리함
  - 함수형 인터페이스 `Function` 을 파라미터로 받는다
- `handle(), handleAsync()`
  - 결과값과 에러를 반환받아 에러가 발생한 경우와 아닌 경우를 모두 처리할 수 있음
  - 함수형 인터페이스 `BiFunction` 을 파라미터로 받는다

```java
@ParameterizedTest
@ValueSource(booleans = {true, false})
void exceptionallyTest(boolean doThrow) throws ExecutionException,
InterruptedException {
        CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
                if (doThrow) {
                        throw new IllegalArgumentException("Invalid Argument");
                }

                return "Thread: " + Thread.currentThread().getName();
        }).exceptionally(e -> {
                return e.getMessage();
        });

        System.out.println(future.get());
}

@ParameterizedTest
@ValueSource(booleans = {true, false})
void handleTest(boolean doThrow) throws ExecutionException, InterruptedException
{
        CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
                if (doThrow) {
                        throw new IllegalArgumentException("Invalid Argument");
                }

                return "Thread: " + Thread.currentThread().getName();
        }).handle((result, e) -> {
                return e == null ? result : e.getMessage();
        });

        System.out.println(future.get());
}
```