Java Thread 를 관리하는 방법

동시성 이슈란?

CPU의 작업 단위인 쓰레드를 여러 개 이용해서 번갈아 작업을 처리하는 방식을 멀티 쓰레드 방식이라고 칭한다.

멀티 쓰레드를 이용하면 공유하는 영역이 많아 멀티 프로세스 방식보다 Context Switching 오버헤드가 작아 메모리 리소스가 상대적으로 적다는 장점이 있다.

하지만 자원을 공유하는 것으로 발생하는 단점이 존재하는데, 그것이 바로 동시성 이슈다.

동시성이란 여러 작업이 독립적으로 실행되는 것을 의미하며, 이와 자주 혼용되는 개념인 병렬성이라는 것이 있는데 병렬성은 동시에 여러 작업이 실행되는 것을 의미한다.

동시성 이슈라는 것은 여러 쓰레드가 하나의 자원을 공유하고 있기 때문에 동일한 자원을 두고 경쟁 상태 같은 문제가 발생한다.

자바의 쓰레드 관리

자바에서는 Thread 를 사용하여 동시성과 병렬 처리를 구현할 수 있다.

자바에서 Thread 를 생성하는 방법은 대표적으로 두 가지 존재하는데, 첫 번째 방법은 `java.lang.Thread` 클래스를 상속받는 방법이고, 두 번째 방법은 `java.lang.Runnable` 인터페이스를 구현하는 방법이다.

```
// Thread 클래스를 상속받는 예제
class JavaThread extends Thread {
        public void run() {
                System.out.println("Thread is Running");
        }
}

public class ThreadExam {
        public static void main(String[] args) {
                JavaThread javaThread = new JavaThread();
                javaThread.start();
        }
}
```

```
class JavaRunnable implements Runnable {
        public void run() {
                System.out.println("Runnable class is Running");
        }
}
```

```
public class RunnableExam {
        public static void main(String[] args) {
                JavaRunnable javaRunnable = new JavaRunnable();
                Thread thread = new Thread(javaRunnable);
                thread.start();
        }
}
```

하지만 위와 같은 방식들로 직접 Thread 와 Runnable 을 사용하는 방식은 다음과 같은 한계점이 존재한다.

- 지나치게 저수준의 API에 의존한다. 여기서 저수준 API는 쓰레드의 생성을 의미한다
  - 쓰레드 생성에 관한 고민은 애플리케이션을 만드는 개발자의 관심과 거리가 멀다
- 값의 반환이 불가능하다
- 매번 쓰레드 생성과 종료를 위한 오버헤드가 발생한다
- 쓰레드들의 관리가 어렵다
  - 직접 쓰레드를 만드는 만큼 쓰레드의 관리가 까다로워진다

위와 같은 단점들을 보완하기 위해 Java는 쓰레드들을 사용하는 방법을 꾸준히 발전시켜오고 있는데,. Java 5에 등장한 **Executor, ExecutorService, ScheduledExecutionService, Callable, Future** 등이 대표적이다.

Callable Interface

기존의 Runnable 인터페이스는 결과를 반환할 수 없다는 한계점이 있었다. 반환값을 얻기 위해서는 번거러운 작업을 거쳐야 했는데 이를 보완하기 위해 Callable 이 추가되었다.

Callable 인터페이스를 구현한 구현체는 제네릭 타입을 사용해 결과를 받을 수 있다.

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

Future Interface

위에서 언급한 Callable 인터페이스의 구현체인 Task는 가용 가능한 쓰레드가 없어서 실행이 미뤄질 수 있고, 작업 시간이 오래 소요될 수 있다.

실행 결과를 바로 받지 못하고 미래의 어느 시점에 얻을 수 있도록 미래에 완료된 Callabe 의 반환값을 구하기 위해 사용되는 것이 Future 인터페이스다.

Future 는 비동기 작업을 가지고 있어 미래에 실행 결과를 얻도록 도와준다. 이를 위해 비동기 작업의 현재 상태를 확인하고 기다리며 결과를 얻는 다양한 방법을 제공한다.

```
public interface Future<V> {

    boolean cancel(boolean mayInterruptIfRunning);

    boolean isCancelled();

    boolean isDone();

    V get() throws InterruptedException, ExecutionException;

    V get(long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException, TimeoutException;
}
```

각각의 메소드들은 다음과 같은 기능을 수행한다

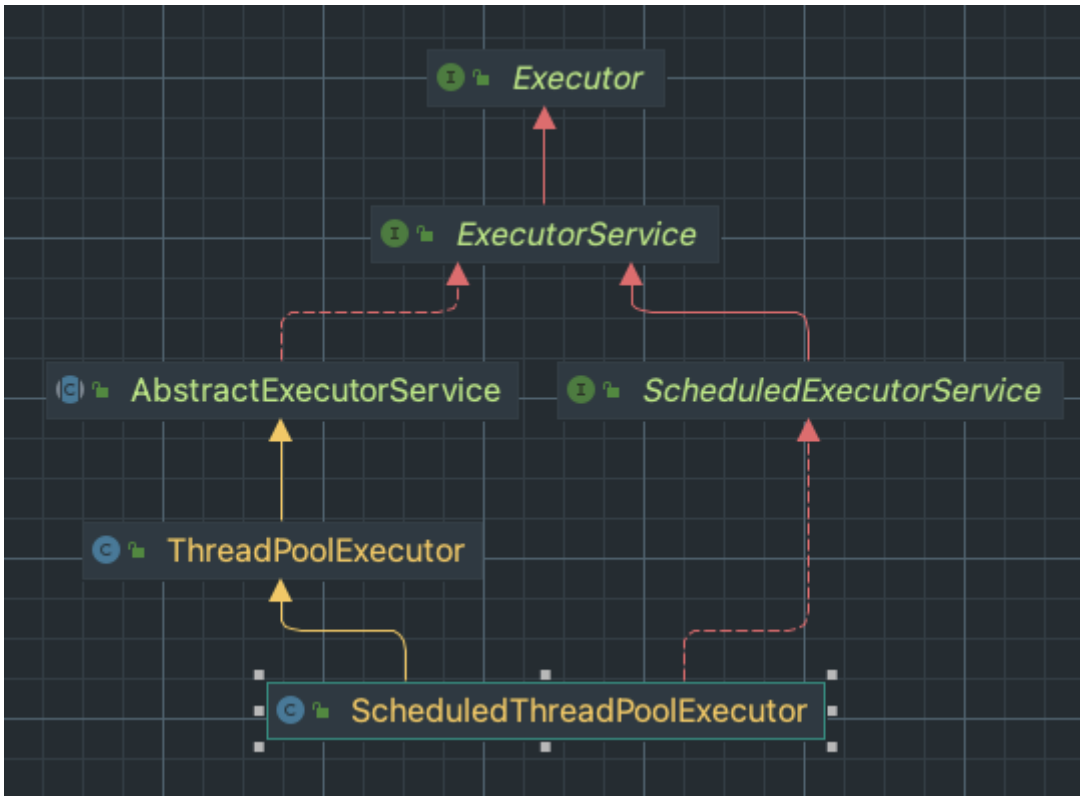
- `get`
  - Blocking 방식으로 결과를 가져온다
  - 타임아웃 설정이 가능하다
- `isDone` / `isCancelled`
  - 각각 작업 완료 여부와 작업 취소 여부를 `boolean` 타입으로 반환한다
- `cancel`
  - 작업을 취소시키며, 취소 여부를 `boolean` 타입으로 반환한다
  - `cancel` 수행 후 `isDone()` 은 항상 `true` 를 반환한다
  - `cancel` 의 파라미터로 `true` 를 전달하면 쓰레드를 `interrupt` 시켜 `InterruptedException` 을 발생시킨다
  - `false` 를 파라미터로 전달하면 진행 중이 작업이 끝날 때까지 대기한다.

```
class JavaFuture {

        private Callabe<String> callabe() {
                return new Callabe<String>() {
                        @Override
                        public String call() throws InterruptedException {
                                Thread.sleep(3000L);
                                return "Thread : " +
Thread.currentThread().getName();
                        }
                }
        }

        @Test
        void get_Test() throws ExecutionException, InterruptedException {
                ExecutorService executorService = new
Executors.newSingleThreadExecutor();
```

```
                Callable<String> callableTask = callabe();

                // Callable 의 구현체의 call() 메소드로 인해 3초간 Blocking
                Future<String> future = executorService.submit(callabeTask);

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

                executorService.shutdown();
        }

        @Test
        void isCancelled_False_Test() {
                ExecutorService executorService = new
Executors.newSingleThreadExecutor();

                Callable<String> callableTask = callabe();

                // Callable 의 구현체의 call() 메소드로 인해 3초간 Blocking
                Future<String> future = executorService.submit(callabeTask);
                assertThat(future.isCancelled()).isFalse();

                executorService.shutdown();
        }

        @Test
        void isCancelled_True_Test() {
                ExecutorService executorService = new
Executors.newSingleThreadExecutor();

                Callable<String> callableTask = callabe();

                // Callable 의 구현체의 call() 메소드로 인해 3초간 Blocking
                Future<String> future = executorService.submit(callabeTask);
                future.cancel(true);
                assertThat(future.isCancelled()).isTrue();

                executorService.shutdown();
        }

        @Test
        void isDone_False_Test() {
                ExecutorService executorService = new
Executors.newSingleThreadExecutor();

                Callable<String> callableTask = callable();

                Future<String> future = executorService.submit(callableTask);
                assertThat(future.isDone()).isFalse();
```

```
                executorService.shutdown();
        }

        @Test
        void isDone_True_Test() {
                ExecutorService executorService = new
Executors.newSingleThreadExecutor();

                Callable<String> callableTask = callable();

                Future<String> future = executorService.submit(callableTask);
                future.cancel(true);
                assertThat(future.isDone()).isTrue();

                executorService.shutdown();
        }

}
```

Executors / Executor / ExecutorService



Java 5 에서 추가된 쓰레드의 생성과 관리를 위한 쓰레드 풀을 위한 기능이 바로 Executor / ExecutorService / ScheduledThreadPoolExecutor 이다.

또한 쓰레드 풀 생성을 돕는 팩토리 클래스인 Executors도 추가되었다.

Executor Interface

동시에 여러 요청을 처리해야 하는 경우에 매번 새로운 쓰레드를 만드는 것은 비효율적이기 때문에, 쓰레드를 미리 만들어두고 재사용하기 위한 Thread Pool 이 등장하게 되었다.

Executor 인터페이스는 쓰레드 풀의 구현을 위한 인터페이스다.

이러한 Executor 인터페이스는 등록된 작업(Runnable) 을 실행하기 위한 인터페이스이며, 작업 등록과 작업 실행 중에서 작업 실행만을 책임진다.

```
public interface Executor {

   void execute(Runnable command);

}
```

```
public class JavaExecutor {

        @Test
        void executor_Run_Test() {
                final Runnable runnable = () -> System.out.println("Thread : " +
Thread.currentThread().getName());

                Executor executor = new RunExecutor();
                executor.execute(runnable);
        }

        static class RunExecutor implements Executor {

                @Override
                public void execute(Runnable command) {
                        new Thread(command).run();
                }

        }

}
```

위의 코드는 전달받은 Runnable 작업을 사용하는 코드를 Executor 로 구현한 코드다. 위 코드는 단순히 객체의 메소드를 호출하는 것이므로, 새로운 쓰레드가 아닌 메인 쓰레드에서 실행된다.

만약 위의 코드를 새로운 쓰레드에서 실행시키려면 아래의 코드처럼 실행시키면 된다.

```
public class JavaExecutor {

        @Test
        void executor_Start_Test() {
                final Runnable runnable = () -> System.out.println("Thread : " +
Thread.currentThread().getName());

                Executor executor = new StartExecutor();
```

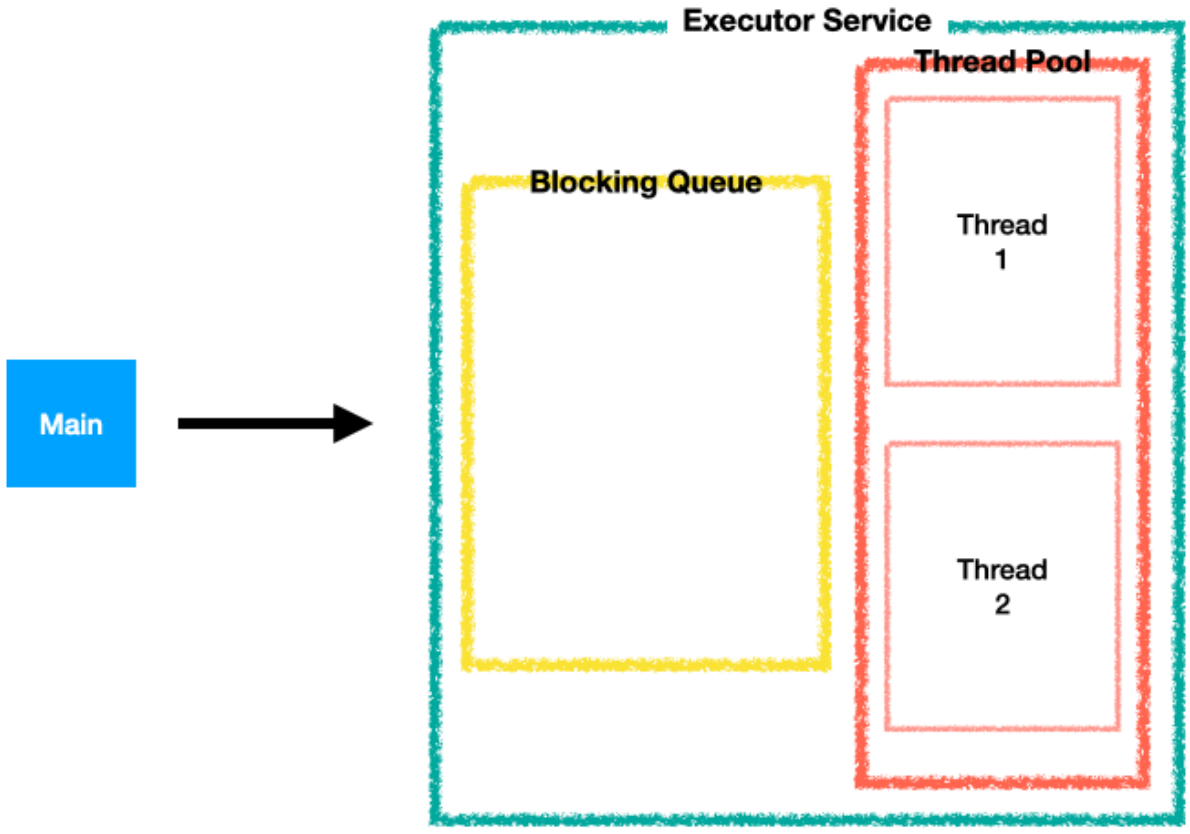
```
                executor.execute(runnable);
        }

        static class StartExecutor implements Executor {

                @Override
                public void execute(Runnable command) {
                        new Thread(command).start();
                }

        }

}
```

ExecutorService Interface

ExecutorService 인터페이스는 작업 (Runnable, Callable) 등록을 위한 인터페이스다.

ExecutorService 는 Executor 를 상속받아서 작업 등록 뿐만 아니라 실행을 위한 책임도 갖는다. 쓰레드 풀은 기본적으로 ExecutorService 인터페이스를 구현한다.

대표적으로 ThreadPoolExecutor 가 ExecutorService 의 구현체인데, ThreadPoolExecutor 내부에 있는 Blocking Queue 에 작업들을 등록해둔다.



위 처럼 크기가 2인 쓰레드 풀이 존재한다고 가정했을 때, 각각의 쓰레드는 작업들을 할당받아 처리한다.

작업이 추가됐을 때, 사용가능한 쓰레드가 없다면 작업은 큐에서 대기하게 된다. 그리고 쓰레드가 작업을 끝내면 다음 작업을 할당받게 되는 것이다.

ExecutorService 는 이를 위해 아래와 같은 기능들을 제공한다.

라이프사이클 관리를 위한 기능

> ExecutorService 는 Executor 의 상태 확인과 적업 종료 등 라이프사이클 관리를 위한 메소드들을 제공한다

- `shutdown`
  - 새로운 작업들을 더 이상 받아들이지 않음
  - `shutdown` 호출 전에 제출된 작업들은 그래도 실행이 끝나고 종료됨
    - Graceful Shutdown
- `shutdownNow`
  - `shutdown` 기능에 더해 이미 제출된 작업들을 인터럽트시킨다
  - 실행을 위해 대기중인 작업 목록을 `List<Runnable>` 형태로 반환한다
- `isShutdown`
  - Executor 의 shutdown 여부를 반환한다
- `isTerminated`
  - `shutdown` 실행 후 모든 작업의 종료 여부를 반환함
- `awaitTermination`
  - `shutdown` 실행 후 지정한 시간 동안 모든 작업이 종료될 때 까지 대기한다
  - 지정한 시간 내에 모든 작업이 종료되었는지 여부를 반환한다

ExecutorService 를 만들어 작업을 실행하면, `shutdown` 이 호출되기 전까지 계속해서 다음 작업을 대기하게 된다.

작업이 완료되었다면 반드시 `shutdown` 을 명시적으로 호출해주어야 한다

아래와 같이 `shutdown` 을 명시하지 않고 메인 메소드를 실행시킬 경우 프로세스가 끝나지 않고 계속해서 다음 작업을 기다리게 된다.

```
public class JavaExecutor {

        public static void main(String[] args) {
                ExecutorService executorService =
Executors.newFixedThreadPool(10);
                Runnable runnable = () -> System.out.println("Thread : " +
Thread.currentThread().getName());
                executorService.execute(runnable);
        }
```

```
}
```

> `shutdown()` 과 `shutdownNow()` 의 차이점
>
> `shutdown()` 은 Thread Pool 에 요청되는 `submit()` 을 더 이상 받아주지 않는 대신, 기존에 실행 중이던 Thread Pool 의 Thread 들은 계속 실행시킨다.
>
> 반대로 `shutdownNow()` 는 Thread Pool 의 모든 Thread 들에게 `thread.interrupt()` 를 실행시켜 하던 작업들을 모두 멈추게 한다.
>
> 하지만 만약 Runnable 혹은 Callable 인터페이스의 구현에 `InturrptedException` 예외를 catch 하여 동작을 멈추는 처리가 없거나, `inturrptFlag` 를 검사하는 코드가 없으면 프로그램이 종료되지 않는다.

```
public class JavaExecutor {

        public static void main(String[] args) {
                Runnable runnable = () -> {
                        System.out.println("Start");
                        while(true) {

                        }
                };

                ExecutorService executorService =
Executors.newFixedThreadPool(10);
                executorService.execute(runnable);
                executorService.shutdownNow();
        }

}
```

위의 코드처럼 Runnable 구현체 내부에 따로 Inturrpt 를 인지할만한 코드가 존재하지 않을 경우, `shutdownNow()` 를 실행해도 Runnable 은 중지하지 않는다.

`shutdownNow()` 를 통해 곧장 Inturrpt 를 반영하고 싶다면 아래의 코드처럼 Inturrpt 를 인식할 수 있는 코드를 추가해주면 된다.

```
public class JavaExecutor {

        public static void main(String[] args) {
                Runnable runnable = () -> {
                        System.out.println("Start");
                        while(true) {
                                if (Thread.currentThread().isInterrupted()) {
                                        System.out.println("This Thread is
Interrupted");
```

```
                                    break;
                                }
                            }
                            System.out.println("End");
                        };

                        ExecutorService executorService =
Executors.newFixedThreadPool(10);
                        executorService.execute(runnable);
                        executorService.shutdownNow();
                }

}
```

추가적으로 `shutdown()` 은 Inturrpt 를 인식할 수 있는 코드의 존재여부와 상관없이 기존에 수행 중이던 Thread 는 멈추지 않고 작동한다.

비동기 작업을 위한 기능들

ExecutorService 는 Runnable 과 Callable 을 작업으로 사용하기 위한 메소드를 제공한다. 동시에 여러 작업들을 실행시키는 메소드도 제공하고 있는데. 비동기 작업의 진행을 추적할 수 있도록 Future 를 반환한다.

반환된 Future 들은 모두 실행된 것이므로 반환괸 `isDone()` 은 항상 `true` 다. 하지만 반드시 정상적으로 종료되지는 않기 때문에 항상 성공한 것은 아니다.

ExecutionService 가 갖는 비동기 작업을 위한 메소드는 아래와 같다.

- `submit`
  - 실행할 작업들을 추가하고 , 작업의 상태와 결과를 포함하는 Future 를 반환함
  - Future 의 get 을 호출하면 성공적으로 작업이 완료된 후 결과를 얻을 수 있다
- `invokeAll`
  - 모든 결과가 나올 때까지 대기하는 블로킹 방식의 요청
  - 최대 Thread Pool 의 크기만큼 작업을 동시에 실행시킨다.
    - 쓰레드가 충분하다면 동시에 실행되는 작업들 중에서 가장 오래 걸리는 작업 만큼 시간이 소요된다
    - 만약 쓰레드가 부족하다면 대기되는 작업들이 발생하므로 가장 오래 걸리는 작업 시간에 더해 추가 시간이 필요하다
  - 동시에 주어진 작업들을 모두 실행하고, 전부 끝나면 각각의 상태와 결과를 갖는 `List<Future>` 을 반환한다
- `invokeAny`
  - 가장 빨리 실행된 결과가 나올 때까지 대기하는 블로킹 방식의 요청
  - 동시에 주어진 작업들을 모두 실행하고. 가장 빨리 완료된 하나의 결과를 Future 로 반환한다

ExecutionService 의 구현체로는 AbstractExecutorService 가 있는데, ExecutorService 의 메소드들에 대한 기본 구현들을 제공한다.

아래는 `invokeAll()` 에 대한 테스트 코드다. 가장 오랜 시간이 걸리는 `sayJava()` 의 시간인 4초만큼의 시간이 소요된다.

```
public class JavaExecutor {

	@Test
	void invokeAll_Test() throws InterruptedException, ExecutionException {
		ExecutorService executorService =
Executors.newFixedThreadPool(10);
		Instant start = Instant.now();

		Callable<String> sayHelloWorld = () -> {
			Thread.sleep(1000L);
			final String result = "Hello World!";
			System.out.println("Result = " + result);
			return result;
		};

		Callable<String> sayJava = () -> {
			Thread.sleep(4000L);
			final String result = "Java!";
			System.out.println("Result = " + result);
			return result;
		};

		Callable<String> sayThread = () -> {
			Thread.sleep(2000L);
			final String result = "Thread!";
			System.out.println("Result = " + result);
			return result;
		};

		List<Future<String>> futures =
executorService.invokeAll(Arrays.asList(sayHelloWorld, sayJava, sayThread));
		for (Future<String> future : futures) {
			System.out.println(future.get());
		}

		System.out.println("time : " + Duration.between(start,
Instant.now()).getSeconds());
		executorService.shutdown();
	}

}
```

`invokeAny()` 의 경우 가장 빨리 끝난 작업 결과만을 구하기 때문에, 동시에 실행된 작업들 중에서 가장 짧게 걸리는 작업만큼 시간이 걸린다.

또한 가장 짧게 걸리는 작업 외의 다른 작업들은 완료 되지 못하므로 `cancel` 처리가 된다. 만약 작업이 진행되는 동안 작업들이 수정되면 결과가 정의되지 않는다.

```
@Test
void invokeAny_Test() throws InterruptedException, ExecutionException {
        ExecutorService executorService = Executors.newFixedThreadPool(10);
        Instant start = Instant.now();

        Callable<String> sayHelloWorld = () -> {
                Thread.sleep(1000L);
                final String result = "Hello World!";
                System.out.println("Result = " + result);
                return result;
        };

        Callable<String> sayJava = () -> {
                Thread.sleep(4000L);
                final String result = "Java!";
                System.out.println("Result = " + result);
                return result;
        };

        Callable<String> sayThread = () -> {
                Thread.sleep(2000L);
                final String result = "Thread!";
                System.out.println("Result = " + result);
                return result;
        };

        String result = executorService.invokeAny(Arrays.asList(sayHelloWorld,
sayJava, sayThread));
        System.out.println(result);

        System.out.println("time : " + Duration.between(start,
Instant.now()).getSeconds());
        executorService.shutdown();
}
```

위의 코드를 실행했을 때 가장 실행시간이 짧은 `sayHelloWorld()` 만큼의 시간이 소요된다.

ScheduledExecutorService Interface

ScheduledExecutorService 는 ExecutorService 를 상속받는 인터페이스로써, 특정 시간 이후에 또는 주기적으로 작업을 실행시키는 메소드가 추가되었다.

특정 시간대에 작업을 실행하거나 주기적으로 작업을 실행하고 싶을 때 사용할 수 있다.

- `schedule`
  - 특정 시간 (delay) 이후에 작업을 실행시킨다
- `scheduleAtFixedRate`
  - 특정 시간 이후 처음 작업을 실행시킨다
  - 작업이 실행되고 특정 시간마다 작업을 실행시킨다
- `scheduleWithFixedDelay`
  - 특정 시간 이후 처음 작업을 실행시킨다
  - 작업이 완료되고 특정 시간이 지나면 작업을 실행시킨다

> `ScheduleAtFixedRate` 와 `ScheduleWithFixedDelay`
>
> `ScheduleAtFixedRate` 는 작업이 실행된 시점부터 delay 가 발생한다.
> `ScheduleWithFixedDelay` 는 작업이 실행된 후 완료된 시점부터 delay 가 발생한다.

```
public class JavaExecutor {

        @Test
        void schedule() throws InterruptedException {
                Runnable runnable = () -> System.out.println("Thread : " +
Thread.currentThread().getName());

                ScheduledExecutorService executorService =
Executors.newSingleThreadScheduledExecutor();

                executorService.schedule(runnable, 1, TimeUnit.SECONDS);
                Thread.sleep(1000L);
                executorService.shutdown();
        }

        @Test
        void schedule_At_Fixed_Rate_Test() throws InterruptedException {
                Runnable runnable = () -> System.out.println("Thread : " +
Thread.currentThread().getName() + " : " + Instant.now());

                ScheduledExecutorService executorService =
Executors.newSingleThreadScheduledExecutor();

                executorService.scheduleAtFixedRate(runnable, 2, 2,
TimeUnit.SECONDS);
                Thread.sleep(10000L);
                executorService.shutdown();
        }

        @Test
        void schedule_With_Fixed_Delay_Test() throws InterruptedException {
```

```
                Runnable runnable = () -> System.out.println("Thread : " +
Thread.currentThread().getName() + " : " + Instant.now());

                ScheduledExecutorService executorService =
Executors.newSingleThreadScheduledExecutor();

                executorService.scheduleWithFixedDelay(runnable, 2, 2,
TimeUnit.SECONDS);
                Thread.sleep(10000L);
                executorService.shutdown();
        }

}
```

Executors

개발자가 직접 쓰레드를 다루는 것은 번거롭고 애플리케이션 개발이라는 목적에도 맞지 않았기 때문에 이를 도와주는 팩토리 클래스인 Executors 가 등장하게 됐다.

Executores 고수준의 동시성 프로그래밍 모델로써 Executor, ExecutorService 또는 ScheduledExecutorService 를 구현한 Thread Pool 을 굉장히 쉽게 생성해준다.

- `newFixedThreadPool`
  - 고정된 쓰레드의 개수를 갖는 Thread Pool 을 생성한다
  - ExecutionService 인터페이스를 구현한 ThreadPoolExecutor 가 생성
- `newCachedThreadPool`
  - 필요할 때 필요한 만큼의 Thread Pool 을 생성한다
  - 이미 생성된 Thread 가 존재한다면 이를 재사용 할 수 있다
- `newScheduledThreadPool`
  - 일정 시간 뒤, 혹은 주기적으로 실행되어야 하는 작업을 위해 Thread Pool 을 생성한다
  - ScheduledExecutorService 인터페이스를 구현한 ScheduledThreadPoolExecutor 객체가 생성된다
- `newSingleThreadExecutor / newSingleThreadScheduledExecutor`
  - 1개의 쓰레드만을 갖는 ThreadPool 을 생성한다

Executors 를 통해 쓰레드의 개수 및 종류를 정할 수 있으며, 이를 통해 쓰레드 생성과 실행 및 관리가 매우 용이해졌다.

Java 5 에서 등장한 Future 는 결과를 얻기 위해서 Blocking 방식으로 대기를 해야 한다는 단점이 존재했기 때문에 이를 보완하여 Java 8 에 추가된 것이 바로 CompletableFuture 다.

> 출처 : https://mangkyu.tistory.com/259