(참고) Thread Pool : 스레드풀

- **스레드풀의 필요성**

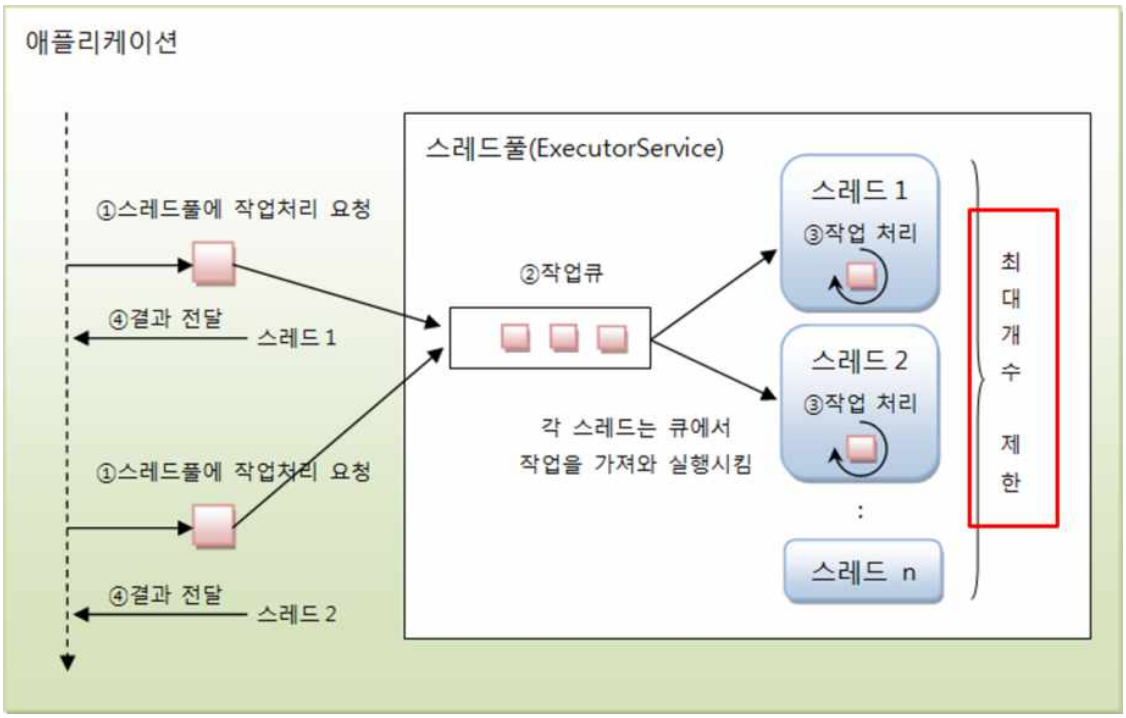
  병렬 작업 처리가 많아지면 스레드 개수가 증가되고 스레드 생성과 스케줄링으로 인해 CPU의 메모리 사용량이 늘어난다. 그로 인해 어플리케이션의 성능이 저하된다. 병렬 작업의 폭증에 따른 스레드 폭증을 막기위해 스레드풀을 사용해야 한다.

- **스레드풀이란?**

  작업 처리에 사용되는 **스레드의 개수를 제한**하면서 쌓인 작업들을 하나씩 맡아 처리하는 방법이다.

  자바는 스레드풀을 생성하고 사용할 수 있도록 java.util.concurrent 패키지에서 ExecutorService 인터페이스와 Executors클래스를 제공한다. Executors의 다양한 정적(static) 메소드를 이용해서 **ExecutorService 구현 객체**를 만들 수 있는데, 이것이 바로 **스레드풀**이다.

- **스레드풀의 동작원리**



1. 스레드풀 생성 및 종료

1.1 스레드풀 생성

Executors 클래스의 다음 두 가지 메소드 중 하나를 이용해서 ExecutorService 구현 객체를 생성할 수 있다.

메소드명(매개 변수)	초기 스레드 수	코어 스레드 수	최대 스레드 수
new**Cashed**ThreadPool( )	**0**	**0**	Integer.MAX.VALUE
new**Fixed**ThreadPool(int i)	**0**	i	i

- 초기 스레드 수 : ExecutorService 객체가 생성될 때 기본적으로 생성되는 스레드 수
- 코드 스레드 수 : 스레드 수가 증가된 후 사용되지 않는 스레드를 스레드풀에서 제거할 때 최소한 유지해야 할 스레드 수
- 최대 스레드 수 : 스레드풀에서 관리하는 최대 스레드 수

< newCashedThreadPool( ) >

```
ExecutorService executorService = Executors.newCashedThreadPool();
```

< newFixedThreadPool(int i) >

```
ExecutorService executorService = Executors.newFixedThreadPool(int i);
```

< 참고) 직접 생성 : 메소드를 사용하지 않고 직접 ThreadPoolExecutor 객체를 생성 >

```
ExecutorService threadPool = new ThreadPoolExecutor(
    3,      // 코어 스레드 수
    100,    // 최대 스레드 수
    120L,   // 놀고 있는 시간
    TimeUnit.SECONDS, //놀고 있는 시간 단위
    new SynchronousQueue<Runnable>() //작업 큐
);
```

1.2 스레드풀 종료

스레드풀의 스레드는 기본적으로 **데몬 스레드가 아니기 때문에** main 스레드가 종료되더라도 작업을 처리하기 위해 계속 실행 상태로 남아있다. 그래서 main() 메소드가 실행이 끝나도 애플리케이션 프로레스는 종료되지 않는다. 애플리케이션이 종료하려면 스레드풀을 종료시켜 스레드들이 종료상태가 되도록 처리해야 한다.

ExecutorService 종료와 관련 메소드

리턴 타입	메소드명(매개 변수)	설명
void	shutdown()	현재 처리 중인 작업을 포함하여 작업 큐에 대기하고 있는 작업을 처리한 뒤에 스레드풀 종료
List(Runnable)	shundownNow()	현재 처리 중인 스레드를 interrupt해서 작업 중지를 시도하고 스레드풀을 종료. 작업 큐에 있는 미처리된 작업(Rannable) 목록을 리턴값으로 반환
boolean	awaitTermination(long timeout, TimeUnit unit)	**shutdown() 메소드 호출 이후**, 모든 작업 처리를 timeout 시간 내에 완료하면 true를반환, 완료하지 못하면 처리 중인 스레드를 interrupt하고 false로 반환

```
executorService.shutdown();

executorService.shutdownNow();
```

2. 작업 생성과 처리 요청

2.1 작업 생성

하나의 작업은 Runnable 또는 Callable 구현 클래스로 표현한다. Runnable의 run( ) 메소드는 리턴값이 없고, Callable의 call( ) 메소드는 리턴값이 있다. 스레드풀의 스레드는 작업 큐에서 Runnable 또는 Callable 객체를 가져와 run( )과 call( ) 메소드를 실행한다.

```
// Runnable 구현 클래스
Runnable task = new Runnable() {
    @Override
    public void run( ){
        // 스레드가 처리할 작업 내용
    }
}
```

```
// Callable 구현 클래스
Callable<T> task = new Callable<T>(){
    @Override
    public T call( ) throws Exception{
        //스레드가 처리할 작업 내용
        return T;
    }
}
```

2.2 작업 처리 요청 by 애플리케이션

작업 처리 요청이란 ExecutorService의 작업 큐에 Runnable 또는 Callable 객체를 넣는 행위이다.

작업 처리 요청 메소드

리턴 타입	메소드명(매개 변수)	설명
void	execute(Runnable command)	- Runnable을 작업 큐에 저장 - 작업 처리 결과를 받지 못함
Future< ? > Future< V > Future< V >	submit(Runnable task) submit(Runnable task, V result) submit(Callable< V > task)	- Runnable 또는 Callable을 작업 큐에 저장 - 리턴된 Future를 통해 작업 처리 결과를 얻을 수 있음

execute ( ) 와 submit ( ) 두 가지 차이점이 있다. 하나는 작업 처리 결과를 반환 여부이다. 나머지는 execute( )는 작업 처리 도중 예외가 발생하면 스레드가 종료되고 해당 스레드는 스레드풀에서 제거된다. 따라서 스레드풀은 다른 작업을 처리하기 위해 새로운 스레드를 생성한다. 반면에 submit( ) 은 작업 처리 도중 예외가 발생하더라도 스레드는 종료되지 않고 다음 작업을 위해 재사용된다. **그렇기 때문에 가급적이면 스레드의 생성 오버헤더를 줄이기 위해 submit( )을 사용하는 것이 좋다.**

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.ThreadPoolExecutor;

public class ExecuteExample {

    public static void main(String[] args) throws Exception {
        //최대 스레드 개수가 2개인 스레드풀 생성
        ExecutorService executorService = Executors.newFixedThreadPool(2);

        for(int i=0; i<10; i++) {
            Runnable runnable = new Runnable() {
                @Override
                public void run() {
                    //스레드 총 개수 및 작업 스레드 이름 출력
                    ThreadPoolExecutor threadPoolExecutor =
(ThreadPoolExecutor)executorService;
                    int poolSize = threadPoolExecutor.getPoolSize();
                    String threadName = Thread.currentThread().getName();
                    System.out.println("[총 스레드 개수: "+poolSize+"] 작업 스레드
이름"+ threadName);

                    //예외 발생 시킴
                    int value = Integer.parseInt("삼");

                }
            };

            //executorService.execute(runnable);
            //executorService.submit(runnable);

            Thread.sleep(500);
        }
    }
}
```

3. 블로킹 방식의 작업 완료 통보

ExecutorService의 submit( )메소드는 매개값으로 준 Runnable 또는 Callable 작업을 스레드풀의 작업 큐에 저장하고 즉시 Future 객체를 리턴한다. Future 객체는 작업 결과가 아니라 작업이 완료될 때까지 기다렸다가(지연했다가 = 블로킹되었다가) 최종 결과를 얻는데 사용된다. 그래서 Future를 **지연 완료(pending completion) 객체**라고 한다. Future의 get( ) 메소드를 호출하면 스레드가 작업을 완료할 때까지 블로킹되었다가 작업을 완료하면 처리 결과를 리턴하다. 이것이 블로킹을 사용하는 작업 완료 통보 방식이다.

리턴 타입	메소드명(매개 변수)	설명
Future<?>	submit(Runnable task)	
Future	submit(Runnable task, V result)	-Runnable 또는 Callable을 작업 큐에 저장
Future	submit(Callable task)	-리턴된 Future를 통해 작업 처리 결과를 얻음

- Future 객체가 가지고 있는 get( ) 메소드

리턴 타입	메소드명(매개 변수)	설명
V	get( )	작업이 완료될 때까지 블로킹되었다가 처리 결과 V를 리턴
V	get(long timeout, TimeUnit unit)	timeout 시간 전에 작업이 완료되면 결과 V를 리턴. 시간 내 완료하지 않으면 TimeoutException을 발생시킨다.

메소드	작업 처리 완료 후 리턴 타입	작업 처리 도중 예외 발생
submit(Runnable task)	future.get( ) - > null	future.get( ) - > 예외 발생
submit(Ruannable task, Integer result)	future.get( ) - > int 타입 값	future.get( ) - > 예외 발생
submit(Callable< String> task)	future.get( ) - > String 타입 값	future.get( ) - > 예외 발생

- 참고(with Java FX)

Future를 이용한 블로킹 방식의 작업 완료 통보에서 주의할 점은 작업을 처리하는 스레드가 작업을 완료하기 전까지는 get( ) 메소드가 블로킹되므로 다른 코드를 실행할 수 없다. 만약 UI를 변경하고 이벤트를 처리하는 스레드가 get( ) 메소드를 호출하면 작업이 완료하기 전까지 UI 를 변경할 수도 없고 이벤트를 처리할 수도 없다. 그렇기 때문에 get( ) 메소드를 호출하는 스레드는 새로운 스레드이거나 스레드풀의 또 다른 스레드가 되어야 한다.

- Future 객체의 작업 결과를 얻기 위한 다른 메소드

리턴 타입	메소드명(매개 변수)	설명
boolean	cancel(boolean myInterruptIfRunning)	작업 처리가 진행 중일 경우 취소시킴
boolean	isCancelled()	작업이 취소되었는지 여부
boolean	isDone()	작업 처리가 완료되었는지 여부

3.1 리턴값이 없는 작업 완료 통보

리턴값이 없는 작업일 경우는 Runnable 객체로 생성한다.

```
//익명 객체
Runnable task = new Runnable(){
  @Override
    public void run(){
        //스레드가 처리할 내용
    }
};
```

결과값이 없는 작업 처리 요청은 submit(Runnable task) 메소드를 이용하면 된다. 결과값이 없음에도 불구하고 다음과 같이 Future 객체를 리턴하는데, 이것은 스레드가 작업 처리를 정상적으로 완료했는지, 아니면 작업 처리 도중 예외가 발생했느지 확인하기 위해서이다.

```
Future future = executorService.submit(task);
```

작업 처리가 정상적으로 완료되었다면 Future의 get()메소드가 null을 리턴하지만 스레드가 작업 처리 도중 interrupt되면 InterruptedException이 발생되고, 예외가 발생하면 ExecutionException이 발생한다. 따라서 예외처리를 해야한다.

```
try {
    future.get();
}catch(InterruptedException e){
    //예외 처리 코드
}catch(ExecutionException e){
    //예외 처리 코드
}
```

<예제 : NoResultExample>

```
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class NoResultExample {

    public static void main(String[] args) {
        ExecutorService executorService = Executors.newFixedThreadPool(
                Runtime.getRuntime().availableProcessors()
        );

        System.out.println("작업 처리 요청");

        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                int sum = 0;
                for(int i=1; i<=10; i++) {sum +=i;}
                System.out.println("처리 결과 : "+sum);
            }
        };

        Future future = executorService.submit(runnable);

            try {
                future.get();
                System.out.println("작업 처리 완료");
            } catch (InterruptedException e) {
                System.out.println("예외 발생"+ e.getMessage());
            } catch (ExecutionException e) {
                System.out.println("예외 발생"+ e.getMessage());
            }
            executorService.shutdown();
    }
}
```

3.2 리턴값이 있는 작업 완료 통보

스레드풀의 스레드가 작업을 완료한 후에 애플리케이션이 처리 결과를 얻어야 한다면 작업 객체를 Callable로 생성해야 한다.

```
Callable<T> task = new Calable<T>(){
    @Override
    public T call() trows Exception{
        //스레드가 처리할 작업 내용
        return T;
    }
};
```

Callable 작업의 처리 요청은 Runnable 작업과 마찬가지로 sumbit 메소드를 호출하면 된다. submit() 메소드는 작업 큐에 Callable 객체를 저장하고 즉시 Future를 리턴한다.

```
Future<T> future = executorService.submit(task);
```

스레드풀의 스레드가 Callable객체의 call( )메소드를 모두 실행하고 T 타입의 값을 반환하면, get() 메소드는 블로킹이 해제되고 T 타입의 값을 리턴하게 된다.

```
try{
    T result = future.get();
}catch(Interrupted Exception e){
 // 예외 처리 코드
}catch(ExecutionException e){
 // 예외 처리 코드
}
```

< 예제 : ResultbyCallableExample >

```
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class ResultCallableExample {

    public static void main(String[] args) {
        ExecutorService executorService = Executors.newFixedThreadPool(
                Runtime.getRuntime().availableProcessors()
        );

        System.out.println("작업 처리 요청");
        Callable<Integer> task = new Callable<Integer>() {
            @Override
            public Integer call() throws Exception {
                int sum = 0;
                for(int i=1; i<=10; i++) {sum +=i;}
                return sum;
            }
        };
        Future<Integer> future = executorService.submit(task);

            try {
                int sum = future.get();
                System.out.println("처리 결과 : "+sum);
                System.out.println("작업 처리 완료");
            } catch (InterruptedException e) {
                System.out.println("예외 발생"+ e.getMessage());
            } catch (ExecutionException e) {
                System.out.println("예외 발생"+ e.getMessage());
            }
            executorService.shutdown();
    }
}
```