제11장 : 스레드를 사용한 병렬처리

Java 실행의 원칙

- 명령을 1개씩 순서대로 실행한다

화면표시 -> 키보드 입력 -> 네트워크 통신 ->

2가지 일을 동시에 하려면??

스레드

Java는 동시에 2가지 이상의 명령을 동시에 수행하기 위해 Thread API를 제공

스레드를 사용하지 않고 표시하는 예


```
for (int i = 0; i < 10; i++) {
    System.out.println(i);
}
```

스레드를 사용하는 예


```java
class PrintingThread extends Thread {
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.print(i);
        }
    }
}
```

```java
public class Main {

    public static void main(String[] args) {
        System.out.println("아무거나 입력");
        Scanner scanner = new Scanner(System.in);

        Thread thread = new PrintingThread();
        thread.start();

        scanner.nextLine();
    }
}
```

Runnable을 이용


```
class PrintingProcess implements Runnable {
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.print(i);
        }
    }
}
```

```
public class Main {

    public static void main(String[] args) {
        System.out.println("아무거나 입력");
        Scanner scanner = new Scanner(System.in);

        Thread thread = new Thread(new PrintingProcess());
        thread.start();

        scanner.nextLine();
    }
}
```

기타 스레드의 여러가지 사용 방법

1. run() : 스레드 시작
2. stop() : 강제로 스레드 정지 (**사용 금지**)
3. suspend() : 일시 정지 (**사용 금지**)
4. destroy() : 파괴 (**사용 금지**)
5. 1개라도 스레드가 남아 있으면 JVM은 종료되지 않음
6. join() : 다른 스레드의 종료를 기다림
7. OS에 따라 동작이 다를 수 있음
8. 예외가 발생해도 메인 스레드에는 영향 없음
9. 여러 스레드가 동시에 1개의 변수를 이용하면 데이터가 망가질 수 있음
10. wait() : 일시적으로 다른 스레드에 양보
11. notifyAll() : 기다리는 스레드를 재개

스레드 세이프한 설계

복수의 스레드를 이용한 프로그래밍을 **멀티스레드 프로그래밍** 이라고 한다.

순서대로 실행되는 것을 **동기**(Synchronization) 이라고 하고
여러 스레드가 동시에 병렬로 실행되는 것을 **비동기**(Asynchronization) 이라고 한다 .

Java에는 여러 스레드를 제어하는 방법을 제공하고 '여러 스레드에서 동시에 이용해도 안전한 클래스나 메서드'는 스**레드 세이프** (Thread safe)한 설계라고 말할 수 있다.

StringBuilder 와 StringBuffer의 차이는 스레드 세이프인지 아닌지이다

synchronized 에 의한 제어

synchronized 로 감싸진 부분을 **실행할 수 있는 것은 1개의 스레드뿐**.
복수의 스레드가 실행할 수 있는 가능성이 있는 부분은 synchronized 블록 내에 작성하면 됨.

```
public class Main {
    int a = 0;
    int b = 0;

    void syncExam() {
        System.out.println("시작");

        synchronized (this) {
            a += 2;
            b = a * 4;
        }

        System.out.println("끝");
    }
```

대상
인스턴스

synchronized 의 다른 사용 방법

메서드 내의 모든 내용을 synchronized 하는 경우 메서드 선언에 synchronized 를 지정하면 동일한 효과를 냄

```java
public class Main {
    int a = 0;
    int b = 0;

    public void syncExam() {
        System.out.println("시작");

        synchronized (this) {
            a += 2;
            b = a * 4;
        }

        System.out.println("끝");
    }

    public synchronized void syncExam2() {
        a += 2;
        b = a * 4;
    }
}
```

synchronized 블록에 의한 방법

synchronized 키워드에 의한 방법

다중 스레드에서 동일 값을 조작하는 코드


```
public class Counter {
    int num = 0;

    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();
        for (int i = 0; i < 1000; i++) {
            new Thread(() -> {
                try {
                    Thread.sleep(10);
                    counter.num += 1;
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }).start();
        }
        Thread.sleep(5000);
        System.out.println(counter.num);
    }
}
```

Quiz : 결과 값은?

동기화를 수행한 코드


```
public class Counter {
    int num = 0;

    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();
        for (int i = 0; i < 1000; i++) {
            new Thread(() -> {
                try {
                    Thread.sleep(10);
                    synchronized (counter) {
                        counter.num += 1;
                    }
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }).start();
        }
        Thread.sleep(5000);
        System.out.println(counter.num);
    }
```

Quiz : 5초 대기 코드를 없애면?

스레드 사용에 있어서의 위험성

잘못 사용하면 데드락(dead lock)에 빠질 수 있음

교착 상태

위키백과, 우리 모두의 백과사전.

이 문서의 내용은 **출처가 분명하지 않습니다.**
이 문서를 편집하여, 신뢰할 수 있는 출처를 표기해 주세요. 검증되지 않은 내용은 삭제될 수도 있습니다. 내용에 대한 의견은 토론 문서에서 나누어 주세요. (2013년 4월)

데드락은 여기로 연결됩니다. 다른 뜻에 대해서는 데드락 (동음이의) 문서를 참조하십시오.

교착 상태(膠着狀態, 영어: deadlock)란 두 개 이상의 작업이 서로 상대방의 작업이 끝나기 만을 기다리고 있기 때문에 결과적으로 아무것도 완료되지 못하는 상태를 가리킨다. 예를 들어 하나의 사다리가 있고, 두 명의 사람이 각각 사다리의 위쪽과 아래쪽에 있다고 가정한다. 이때 아래에 있는 사람은 위로 올라 가려고 하고, 위에 있는 사람은 아래로 내려오려고 한다면, 두 사람은 서로 상대방이 사다리에서 비켜줄 때까지 하염없이 기다리고 있을 것이고 결과적으로 아무도 사다리를 내려오거나 올라가지 못하게 되듯이, 전산학에서 교착 상태란 다중 프로그래밍 환경에서 흔히 발생할 수 있는 문제이다. 이 문제를 해결하는 일반적인 방법은 아직 없는 상태이다.

데드락이 발생하는 코드 예시


```
public class DeadlockExample {
   public static void main(String[] args) {
      final Object resource1 = "resource1";
      final Object resource2 = "resource2";

      Thread thread1 = new Thread(() -> {
         synchronized (resource1) {
            System.out.println("Thread 1: locked resource 1");
            try { Thread.sleep(100);} catch (InterruptedException e) {}
            synchronized (resource2) {
               System.out.println("Thread 1: locked resource 2");
            }
         }
      });

      Thread thread2 = new Thread(() -> {
         synchronized (resource2) {
            System.out.println("Thread 2: locked resource 2");
            try { Thread.sleep(100);} catch (InterruptedException e) {}
            synchronized (resource1) {
               System.out.println("Thread 2: locked resource 1");
            }
         }
      });

      thread1.start();
      thread2.start();
   }
}
```

thread1 은 resource1 을 잠그고
resource2 또한 잠그려고 하고

thread2 는 resource2 를 잠그고
resource1 또한 잠그려고 합니다.

결국 닭이 먼저냐 달걀이 먼저냐로 둘 다
잠긴 상태로 아무것도 수행을 못 하는
상태에 빠짐

OK. 이제 스레드를 사용해 보자!!

스레드 사용시 고려할 점

- 되도록 스레드를 사용하지 않는 방법을 생각한다
- 어쩔 수 없는 경우에만 스레드를 사용하고, 리스크를 감수할 각오를 하고, 개발기간이나 테스트에 충분한 시간을 가진다.

- **테스트나 문제 해결이 무지하게 어렵다.**
- **개발 공정 전체에 많은 영향을 끼칠 수 있다.**

- 스레드 등 병렬처리(비동기)는 고도의 스킬이 필요하다.

Concurrency Utilities 의 이용

스레드에 대한 고수준 API

- Executor
  - Thread를 직접 사용하지 않고 병렬 처리가 가능
  - 처리 효율을 높일 수 있는 스레드풀(thread pool) 등을 이용 가능

- 스레드 세이프한 컬렉션
  - ConcurrentHashMap 등 기존과 동일한 기능 + 스레드 세이프한 클래스를 제공

- Synchronized 또는 Lock 에 관한 수행
  - 복수 스레드를 잘 사용하기 위한 기능을 제공하는 클래스들 다수 제공
  - CountDownLatch, CyclicBarrier, Exchanger, Semaphore 등

- **스레드 풀**은 큐에서 작업을 동시에 실행하는 관리형 스레드 컬렉션입니다. 새 작업은 기존 스레드가 유휴 상태가 되면 이 스레드에서 실행됩니다. 작업을 스레드 풀로 보내려면 **ExecutorService** 인터페이스를 사용합니다.
-
- 다음 예에서는 **4**개의 스레드로 구성된 스레드 풀을 만듭니다.

```
ExecutorService executorService = Executors.newFixedThreadPool(4);
```

(중급) 백그라운드 스레드에서 실행

먼저 백그라운드 응답을 처리할 Result 클래스입니다

```java
// Result.java
public abstract class Result<T> {
    private Result() {}

    public static final class Success<T> extends Result<T> {
        public T data;

        public Success(T data) {
            this.data = data;
        }
    }

    public static final class Error<T> extends Result<T> {
        public Exception exception;

        public Error(Exception exception) {
            this.exception = exception;
        }
    }
}
```

(중급) 네트워크 요청의 스레드 차단

일반적으로 네트워크 요청은 스레드를 차단합니다

```
public class LoginRepository {

    private final String loginUrl = "https://example.com/login";
    private final LoginResponseParser responseParser;

    public LoginRepository(LoginResponseParser responseParser) {
        this.responseParser = responseParser;
    }

    public Result<LoginResponse> makeLoginRequest(String jsonBody) {
        try {
            URL url = new URL(loginUrl);
            HttpURLConnection httpConnection = (HttpURLConnection) url.openConnection();
            httpConnection.setRequestMethod("POST");
            httpConnection.setRequestProperty("Content-Type", "application/json; charset=utf-8");
            httpConnection.setRequestProperty("Accept", "application/json");
            httpConnection.setDoOutput(true);
            httpConnection.getOutputStream().write(jsonBody.getBytes("utf-8"));

            LoginResponse loginResponse = responseParser.parse(httpConnection.getInputStream());
            return new Result.Success<LoginResponse>(loginResponse);
        } catch (Exception e) {
            return new Result.Error<LoginResponse>(e);
        }
    }
}
```

(중급) Executor 인터페이스를 활용하기 위한 준비

인스턴스화한 스레드 풀을 사용하여 실행을 백그라운드 스레드로 이동할 수 있습니다. 먼저 종속 항목 삽입 원칙에 따라 LoginRepository는 코드를 실행하지만 스레드를 관리하지 않기 때문에 ExecutorService와는 대조적으로 Executor의 인스턴스를 사용합니다.

```
public class LoginRepository {
    ...
    private final Executor executor;

    public LoginRepository(LoginResponseParser responseParser, Executor executor) {
        this.responseParser = responseParser;
        this.executor = executor;
    }
    ...
}
```

(중급) 비동기 호출

Executor의 execute() 메서드는 Runnable을 사용합니다. Runnable은 호출 시 스레드에서 실행되는 run() 메서드가 있는 단일 추상 메서드(SAM) 인터페이스입니다.

```
public class LoginRepository {
    ...
    public void makeLoginRequest(final String jsonBody) {
        executor.execute(new Runnable() {
            @Override
            public void run() {
                Result<LoginResponse> ignoredResponse = makeSynchronousLoginRequest(jsonBody);
            }
        });
    }

    public Result<LoginResponse> makeSynchronousLoginRequest(String jsonBody) {
        ... // HttpURLConnection logic
    }
    ...
}
```

(중급) 기본 스레드와의 통신

makeLoginRequest() 함수는 값을 비동기적으로 반환할 수 있도록 콜백을 매개변수로 사용해야 합니다.
결과가 포함된 콜백은 네트워크 요청이 완료되거나 실패가 발생할 때마다 호출됩니다.
자바에서는 콜백 인터페이스를 만들어야 합니다.

```
interface RepositoryCallback<T> {
    void onComplete(Result<T> result);
}

public class LoginRepository {
    ...
    public void makeLoginRequest(
        final String jsonBody,
        final RepositoryCallback<LoginResponse> callback
    ) {
        executor.execute(new Runnable() {
            @Override
            public void run() {
                try {
                    Result<LoginResponse> result = makeSynchronousLoginRequest(jsonBody);
                    callback.onComplete(result);
                } catch (Exception e) {
                    Result<LoginResponse> errorResult = new Result.Error<>(e);
                    callback.onComplete(errorResult);
                }
            }
        });
    }
  ...
}
```

(중급) 콜백을 구현


```
String jsonBody = "{ username: \"" + username + "\", token: \"" + token + "\" }";
loginRepository.makeLoginRequest(jsonBody, new RepositoryCallback<LoginResponse>() {
    @Override
    public void onComplete(Result<LoginResponse> result) {
        if (result instanceof Result.Success) {
            // Happy path
        } else {
            // Show error in UI
        }
    }
});
```

연습문제 10-1

0~50까지의 정수를 출력하는 CountUpThread 클래스를 작성하시오.

작성한 CountUpThread 인스턴스 3개를 생성하고 실행하시오.

연습문제 10-2

다음 클래스는 여러 스레드로부터 동시에 이용될 때 데이터가 잘못될 수 있습니다.
스레드에 안전한 코드로 수정하시오.

```
public class Counter {
  private long count = 0;

  public void add(long i) {
    System.out.println("더하기");
    count += i;
  }

  public void mul(long i) {
    System.out.println("곱하기");
    count *= i;
  }
}
```

이후의 학습 내용

1. 외부 라이브러리 사용
2. javadoc 으로 클래스의 문서화
3. javac, java, jar 커맨드
4. 유닛 테스트
5. 코드 품질과 리팩토링
6. 팀에 의한 개발과 코드의 공유 (git)
7. 애자일 개발, 스크럼, 빌드 자동화
8. 설계 원칙, 디자인 패턴
9. 좋은 품질의 코드 작성