


이펙티브 자바 CP.10

🕒 작성 일시	@2023년 3월 25일 오후 10:04
🕒 최종 편집 일시	@2023년 3월 26일 오전 2:12
📄 유형	이펙티브 자바
👤 작성자	 종현 박
👥 참석자	
🗣️ 언어	

10 동시성

- 78. 공유 중인 가변 동기화해 사용하라
- 79. 과도한 동기화는 피하라
- 80. 스레드보다는 실행자, 태스크, 스트림을 애용하라
- 81. wait 와 notify 보다는 동시성 유틸리티를 애용하라
- 82. 스레드 안전성 수준을 문서화 해라
- 83. 지연 초기화는 신중히 사용하
- 84. 프로그램의 동작을 스레드 스케줄러에 기대지 말라

10 동시성

▼ 78. 공유 중인 가변 동기화해 사용하라

- `synchronized` 키워드 (동기화)
 - 해당 메서드나 블록을 한번에 한 스레드씩 수행하도록 보장한다.
 - 동기화를 배타적 실행, 즉 한 스레드가 변경하는 중이라서 상태가 일관되지 않은 순간의 객체를 다른 스레드가 보지 못하게 막는 용도.
 - 한 객체가 일관된 상태를 가지고 생성되고 (아이템 17), 이 객체에 접근하는 메서드는 그 객체에 락을 건다. 락을 건 메서드는 객체의 상태를 확인하고 필요하면 수정한다. 즉, 객체를 하나의 일관된 상태에서 다른 일관된 상태로 변화시킨다. 동기화를 제대로 사용하면 어떤 메서드도 이 객체의 상태가 일관되지 않는 순간을 볼 수 없을 것이다.

- 동기화 없이는 한 스레드가 만든 변화를 다른 스레드에서 확인하지 못할 수 있다. 동기화는 일관성이 깨진 상태를 볼 수 없게하는 것은 물론, 동기화된 메서드나 블록에 들어간 스레드가 같은 락의 보호하에 수행된 모든 이전 수정의 최종 결과를 보게 해준다.

- 원자성

- 언어 명세상 `long`, `double` 외의 변수를 읽고 쓰는 동작은 원자적이라고 한다.
- 여러 스레드가 같은 변수를 동기화 없이 수정하는 중이라도, 항상 어떤 스레드가 정상적으로 저장한 값을 온전히 읽어옴을 보장하는 뜻이다.
- 하지만, 원자적 데이터를 읽고 쓸때는 동기화를 하지 말아야 겠다는 생각은 아주 위험한 생각이다. 자바 언어 명세는 스레드가 필드를 읽을 때 항상 수정이 완전히 반영된 값을 얻는다고 보장하지만, 한 스레드가 저장한 값이 다른 스레드에게 보이는지는 보장하지 않는다.
 - 이는 한 스레드가 만든 변화가 다른 스레드에게 언제 어떻게 보이는지를 규정한 자바의 메모리 모델 때문
- 동기화는 배터적 실행뿐 아니라, 스레드 사이의 안정적인 통신에 꼭 필요하다.

- 동기화 실패

- 공유 중인 가변 데이터를 비록 원자적으로 읽고 쓸 수 있을지라도 동기화에 실패하면 처참한 결과로 이어질 수 있다.
- `Thread.stop` 데이터가 훼손 될 수 있다. (사용 금지) - deprecated 되었다. (1.2)

- 올바르게 스레드를 멈추는 코드

- 첫 번째 스레드는 자신의 `boolean` 필드를 폴링하면서 그 값이 `true` 가 되면 멈춘다.
- 이 필드를 `false` 로 초기화 해두고, 다른 스레드에서 이 스레드를 멈추고자 할 때 `true` 로 변경하는 식이다.
- ex) 잘못된 방식 - `boolean` 필드가 원자적이라 동기화를 안 한 경우

```
public class StopThread {
    private static boolean stopRequested;

    public static void main(String [] args) throw InterruptedException {
        Thread backgroundThread = new Thread(() -> {
            int i = 0;
            while (!stopRequested)
                i++;
        });
        backgroundThread.start();
    }
}
```

```

        TimeUnit.SECONDS.sleep(1);
        stopRequested = true;
    }
}

```

- 메인 스레드가 1초 후 `stopRequested` 를 `true` 로 설정하는 `backgroundThread` 는 반복문을 빠져나올 것 처럼 보이지만, 영원히 수행된다.
- 동기화하지 않으면 메인 스레드가 수정한 값을 백그라운드 스레드가 언제쯤에나 보게될지 보장할 수 없다.
- 호이스팅 (끌어올리기)
 - OpenJDK 서버 VM 이 실제로 호이스팅이라는 최적화 기법으로 해당 코드는 이렇게 변경된다.

```

// before
while (!stopRequest)
    i++;

// after
if (!stopRequested)
    while (true)
        i++;

```

- 즉, 이 프로그램은 응답 불가 상태가 되어 더 이상 진전이 없다.

◦ ex) 올바른 방법 `synchronized` 사용

```

public class StopThread {
    private static boolean stopRequested;

    private static synchronized void requestStop() {
        stopRequested = true;
    }

    private static synchronized boolean stopRequested() {
        return stopRequested;
    }

    public static void main(String [] args) throw InterruptedException {
        Thread backgroundThread = new Thread(() -> {
            int i = 0;
            while (! stopRequested())
                i++;
        });
        backgroundThread.start();
    }
}

```

```

        TimeUnit.SECONDS.sleep(1);
        requestStop();
    }
}

```

- 쓰기, 읽기 모두 동기화되지 않으면 동작을 보장하지 않는다. 반드시 쓰기, 읽기를 `synchronized` 통해 가변 데이터를 동기화 하자. (통신 목적으로 사용된 것)
- ex) 올바르게 더 속도가 빠른 대안 `volatile` 선언해서 동기화

```

public class StopThread {
    private static volatile boolean stopRequested;

    public static void main(String [] args) throw InterruptedException {
        Thread backgroundThread = new Thread(() -> {
            int i = 0;
            while (!stopRequested)
                i++;
        });
        backgroundThread.start();

        TimeUnit.SECONDS.sleep(1);
        stopRequested = true;
    }
}

```

- **`volatile` 한정자 (통신 목적)**

- 배터적 수행과는 상관 없지만 항상 가장 최근에 기록된 값을 읽게 됨을 보장한다.
- 하나의 스레드에서만 쓰기 작업, 나머지 여러 스레드에서 읽기 작업을 보장한다
- 하지만, 주의해서 사용해야한다.

- ex) 일련번호 생성

```

private static volatile int nextSerialNumber = 0;

public static int generateSerialNumber() {
    return nextSerialNumber++;
}

```

- 이 메서드는 매번 고유한 값을 반환할 의도로 만들어졌다.
- 이 메서드의 상태는 `nextSerialNumber` 라는 필드로 결정되는데, 원자적으로 접근할 수 있고 어떤 값이든 허용한다. 따라서 굳이 동기화하지

않아도 불변식을 보호할 수 있어 보인다. (하지만, 이 역시 동기화 없이는 올바르게 동작하지 않는다.)

- 문제는 증가 연산자(++) 다.
이 연산자는 코드상으로는 하나지만, 실제로는 `nextSerialNumber` 에 두 번 접근 한다. 먼저 값을 읽고, 그런 다음 새로운 값을 저장하는 것이다.
- 만약 두 번째 스레드가 이 두 접근 사이를 비집고 들어와 값을 읽어가면 첫 번째 스레드와 똑같은 값을 돌려받게 된다. 이런 오류를 안전 실패라고 한다.
 - 이런 문제를 해결하기 위해 `synchronized` 한정자를 붙이고 `volatile` 을 제거해야한다.

- **atomic 패키지**

- 동시성 프로그래밍을 위한 여러 자바 표준 기술들을 제공하는데 락 없이도 스레드 안전한 프로그래밍들을 지원하는 클래스들이 다양하게 존재하고, `volatile` 과 다르게 안전 통신 목적 뿐만 아니라, 배타적 실행 모두 제공하면서 성능도 좋다
- ex) `AtomicLong` 사용

```
private static final AtomicLong nextSerialNum = new AtomicLong();

public static long generateSerialNumber() {
    return nextSerialNum.getAndIncrement();
}
```

- **가장 좋은 방법은 애초에 가변 데이터를 공유하지 않는 것이다.**

- 불변 데이터 (아이템 17)만 공유하거나 아무것도 공유하지 말자.
가변 데이터는 단일 스레드에서만 쓰도록 하자.
- 한 스레드가 데이터를 다 수정한 후 다른 스레드에 공유할 때는 해당 객체에서 공유하는 부분만 동기화해도 된다. 그러면 그 객체를 다시 수정할 일이 생기기 전까지 다른 스레드들은 동기화 없이 자유롭게 값을 읽어갈 수 있다. 이런 객체를 사실상 **불변** 이라 하고 다른 스레드에 이런 객체를 건네는 행위를 **안전 발행** 이라 한다.
 - 객체를 안전하게 발행하는 방법으로는 초기화 과정에서 객체를 정적 필드, `volatile` 필드, `final` 필드, 보통의 `lock` 을 통해 접근 하는 필드에 저장 등이 있다.

- 정리

- 여러 스레드가 가변 데이터를 공유하면 그 데이터를 읽고 쓰는 동작은 반드시 동기화해야한다.
- `synchronized` , `atomic` 패키지 을 사용하면, 배타적 실행과 안전한 통신을 가능하게 할 수 있다.
- `volatile` 을 사용하면 안전한 통신만 가능하게 할 수 있다.

▼ 79. 과도한 동기화는 피하라

▼ 80. 스레드보다는 실행자, 태스크, 스트림을 애용하라

▼ 81. wait 와 notify 보다는 동시성 유틸리티를 애용하라

▼ 82. 스레드 안전성 수준을 문서화 해라

▼ 83. 지연 초기화는 신중히 사용하

▼ 84. 프로그램의 동작을 스레드 스케줄러에 기대지 말라