
이펙티브 자바

item82 ~ item84

24/07/30

Item 82

스레드 안전성 수준을 문서화하라

아이템 82 스레드 안전성 수준을 문서화하라

한 메서드를 여러 스레드가 동시 호출할 때는 레이스 조건이 발생하면 안되고,

데이터 무결성이 깨지지 않아야 한다.

API 문서를 봤을 때 `synchronized` 한정자가 보이면 멀티 스레드가 안전하다고 하지만,
이것만으로는 안전하다고 믿기 어렵고, 스레드 안전성 수준에서도 갈릴 수 있다.

아이템 82 스레드 안전성 수준을 문서화하라

- 불변: 이 클래스의 인스턴스는 마치 상수와 같아 외부 동기화도 필요x
 - String, Long, BigInteger 등등
- 무조건적 스레드 안전: 이 클래스의 인스턴스는 수정될 수 있으나, 내부에서 충실히 동기화하여 별도의 외부 동기화 없이 사용해도 안전하다.
 - AtomicLong, ConcurrentHashMap 등
- 조건부 스레드 안전: 무조건적 스레드 안전과 같으나, 일부 메서드는 동시에 사용하려면 외부 동기화가 필요하다.
 - Collections.synchronized 래퍼 메서드가 반환한 컬렉션들
- 스레드 안전하지 않음: 이 클래스의 인스턴스는 수정될 수 있고, 동시에 사용하려면 각각의 메서드 호출을 클라이언트가 선택한 외부 동기화 메커니즘으로 감싸야 한다.
 - ArrayList, HashMap 등
- 스레드 적대적: 이 클래스는 모든 메서드 호출을 외부 동기화로 감싸도 멀티스레드 환경에서 안전하지 않다. 보통 deprecated API로 지정된다.

스레드 안전성



아이템 82 스레드 안전성 수준을 문서화하라

외부에서 사용할 수 있는 락을 제공할 때

이 때 클라이언트에서는 일련의 메서드 호출을 원자적으로 수행할 수 있다.

다만, 클래스가 공개된 락을 오래 쥐고 놓지 않는 DOS 공격을 수행할 수도 있기 때문에, synchronized 메서드 대신 비공개 락 객체를 사용하자.

아이템 82 스레드 안전성 수준을 문서화하라

Dos 공격의 종류

버퍼 오버플로 공격

메모리 버퍼 오버플로로 인해 컴퓨터가 사용 가능한 모든 하드 디스크 공간, 메모리, CPU 시간을 소모하도록 만들 수 있는 공격 유형이 있다.

이러한 형태의 악용은 느린 동작, 시스템 충돌, 기타 유해한 서버 동작을 초래하여 서비스 거부를 초래하는 경우가 많다.

폭주

대상 서버를 압도적인 양의 패킷으로 포화시킴으로써 서버 용량을 과포화시켜 서비스 거부를 초래할 수 있다. 대부분의 DoS 폭주 공격이 성공하려면 악의적인 행위자가 대상보다 더 많은 가용 대역폭을 가지고 있어야 한다.

추가로, 비공개 락 객체 관용구는 무조건적 스레드 안전 클래스에서만 사용할 수 있다.

➡ 조건부 스레드 안전 클래스에서는 특정 호출 순서에 필요한 락이 무엇인지 클라이언트에게 알려줘야 하므로.

아이템 82 스레드 안전성 수준을 문서화하라

NOTE

결론

모든 클래스는 자신의 스레드 안전성 정보를 명확히 문서화해야 한다.

synchronized 한정자는 문서화와는 관련이 없다.

조건부 스레드 안전 클래스는 메서드를 어떤 순서로 호출할 때 외부 동기화가 요구되고, 그 때 어떤 락을 얻어야 하는지도 알려줘야 한다

Item 83

지연 초기화는 신중히 사용하라

아이템 83 지연 초기화는 신중히 사용하라

지연 초기화(lazy initialization)란?

➡ 필드의 초기화 시점을 그 값이 처음 필요할 때까지 늦추는 기법

사용 이유

➡ 주로 최적화 용도 혹은 클래스와 인스턴스 초기화 시 발생하는 순환 문제를 해결하는 효과가 있다.

지연초기화의 필요성

만약 필드를 초기화하는 비용은 크고, 필드를 사용하는 인스턴스의 비율이 낮을 때는 지연 초기화가 필요하다.

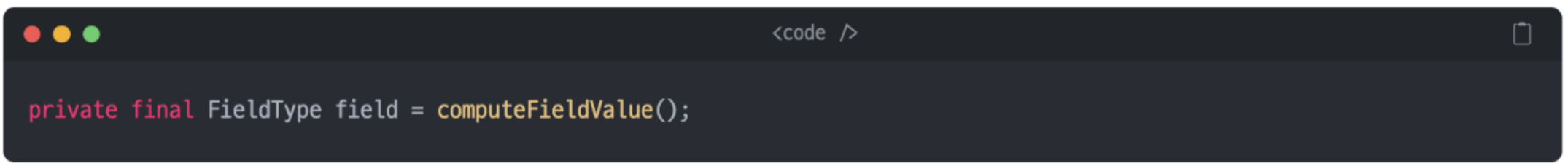
➡ 이를 알 수 있는 방법은 지연 초기화 적용 전후의 성능을 측정해보는 것이다.

아이템 83 지연 초기화는 신중히 사용하라

지연초기화의 단점

지연초기화를 사용하면 클래스나 인스턴스 생성 시 초기화 비용은 줄지만,
해당 필드에 접근하는 비용이 커진다.
따라서 초기화 된 각 필드를 얼마나 빈번히 호출하느냐에 따라 실제로는
성능을 느려지게 할 수도 있다.

인스턴스 필드를 초기화하는 일반적인 방법

A code editor window with a dark background. It has three colored window control buttons (red, yellow, green) in the top-left corner. The title bar contains the text "<code />" and a close button icon in the top-right corner. The code area contains a single line of Java code: `private final FieldType field = computeFieldValue();`

```
private final FieldType field = computeFieldValue();
```

아이템 83 지연 초기화는 신중히 사용하라

인스턴스 필드의 지연 초기화 - synchronized 접근자 방식

```
<code />

private FieldType field;

private synchronized FieldType getField() {
    if (field == null)
        field = computeFieldValue();
    return field;
}
```

아이템 83 지연 초기화는 신중히 사용하라

정적 필드 지연 초기화

```
public class Main {
    public static void main(String[] args) {

        FieldHolder.getField();
        FieldHolder.getField();
        FieldHolder.getField();
    }

    private static class FieldHolder {
        static final FieldType field = computeFieldValue();

        private static FieldType computeFieldValue() {
            System.out.println("create FieldType");
            return new FieldType();
        }

        private static FieldType getField() {
            return FieldHolder.field;
        }
    }
}
```

아이템 83 지연 초기화는 신중히 사용하라

인스턴스 필드 지연 초기화용 이중검사 관용구

```
private volatile FieldType field;

private FieldType getField() {
    FieldType result = field;
    if (result != null) { //첫 번째 검사 (락 사용 안 함)
        return result;
    }

    synchronized(this) {
        if (field == null) { //두 번째 검사 (락 사용)
            field = computeFieldValue();
        }
        return field;
    }
}
```

위의 경우에서 반복해서 초기화해도 상관없는 인스턴스 필드를 지연초기화 하는 경우라면, 이중 검사에서 두 번째 검사를 생략해도 괜찮다.
(단, 필드는 여전히 volatile을 사용할 것.)

아이템 83 지연 초기화는 신중히 사용하라

NOTE 결론

대부분의 필드는 지연시키지 말고 곧바로 초기화하자. 만약 성능 혹은 초기화 순환을 막기 위해 꼭 지연 초기화를 써야 한다면 올바른 지연 초기화 기법을 사용하자.
인스턴스 필드 ➡ 이중 검사 관용구정적 필드 ➡ 지연 초기화 홀더 클래스 관용구

Item 84

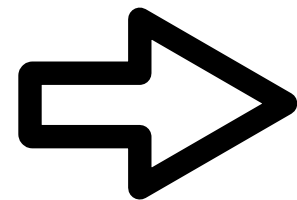
프로그램의 동작을 스레드 스케줄러에 기대지 말라

아이템 84 프로그램의 동작을 스레드 스케줄러에 기대지 말라

정상적인 운영체제라면 이 작업을 공정하게 수행하지만, 구체적인 스케줄링 정책은 OS마다 다를 수 있다.

또한 실행 가능한 스레드의 평균적인 수를 프로세서 수보다 지나치게 많아지지 않도록 하는 것이다.

그럼 적정 스레드, 스레드 풀 개수는?



적정 스레드 개수 = cpu 수 * (1+ 대기 시간/서비스 시간)

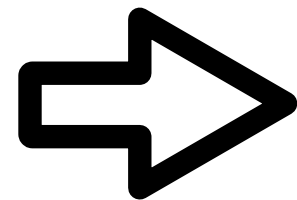
적정 스레드 풀 개수 = CPU 수 * (CPU 목표 사용량) * (1+대기 시간/서비스 시간)

아이템 84 프로그램의 동작을 스레드 스케줄러에 기대지 말라

정상적인 운영체제라면 이 작업을 공정하게 수행하지만, 구체적인 스케줄링 정책은 OS마다 다를 수 있다.

또한 실행 가능한 스레드의 평균적인 수를 프로세서 수보다 지나치게 많아지지 않도록 하는 것이다.

그럼 적정 스레드, 스레드 풀 개수는?



적정 스레드 개수 = cpu 수 * (1+ 대기 시간/서비스 시간)

적정 스레드 풀 개수 = CPU 수 * (CPU 목표 사용량) * (1+대기 시간/서비스 시간)

ex) Worker 스레드는 요청에 대한 응답을 JSON으로 변환하고 몇 가지 규칙을 실행하는 microservice를 호출한다 가정하자.
응답 시간은 50ms, 서비스 시간은 5ms이며 worker 스레드를 실행시키는 프로그램은 코어가 두 개인 cpu라 가정한다면
적절 스레드 풀 사이즈는 $2 * (1 + 50/5) = 22$ 가된다.

아이템 84 프로그램의 동작을 스레드 스케줄러에 기대지 말라

스레드의 우선 순위?

스레드 우선순위는 자바에서 이식성이 가장 나쁜 특성에 속한다.

NOTE

스레드 우선순위란 대기하고 있는 상황에 더 먼저 수행할 수 있는 순위를 말한다.

우선순위 값은 스레드 간 관계를 잘 파악하고 있는 상태에서 변경해야 장애가 발생하지 않는다.

스레드 몇 개의 우선순위를 조율해서 애플리케이션의 반응 속도를 높이는 것도 일견 타당할 수 있으나,

정말 그래야 할 상황은 드물고 이식성이 떨어진다.

심각한 응답 불가 문제를 스레드 우선순위로 해결하려는 시도는 절대 합리적이지 않다.

아이템 84 프로그램의 동작을 스레드 스케줄러에 기대지 말라

NOTE 결론

프로그램의 동작을 스레드 스케줄러에 기대지 말자.

Thread.yield와 스레드 우선순위에 의존하지도 말자.