

# 스레드 안전성

## 2.1 스레드 안전성이란?

안정적인 동시성 프로그램을 만들기 위해 **스레드와 락**을 적절히 사용해야 합니다.

- 스레드 안전한 코드를 작성한다는 것은, **프로그램의 상태를 공유되고 변경 가능한 상태에 대한 접근을 통제**하는 일입니다.
- **공유**란 여러 스레드가 해당 변수에 접근할 수 있다는 것을 의미합니다.

### 2.1.1 예제 :상태 없는 서블릿

프레임워크가 만들어낸 스레드 안에서 내 컴포넌트가 호출될 때, 내가 **만든 코드가 스레드 안전**해야 합니다.

```
//스레드를 사용하는 프레임워크 서블릿
//안전한 스레드를 위해선 필드도 없고 다른 클래스의 필드도 참조하면 안됨
@ThreadSafe
public class StatelessFactorizer implements Servlet {
    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        encodeIntoResponse(resp, factors);
    }
}
```

- 지역 변수만 사용 = 공유 상태 없음
- 공유 상태가 없으면 동기화도 필요 없음
- 즉 스레드 간 충돌의 여지가 없음 → 안전

무상태로 쓰자

## 2.2 단일 연산

“상태 없는 서블릿은 안전했는데, 거기에 상태 하나만 추가하면 뭐가 문제인데?”

```
@NotThreadSafe
public class UnsafeCountingFactorizer implements Servlet {
    private long count = 0;

    public long getCount() { return count; }

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        ++count; // 이게 문제다!
        encodeIntoResponse(resp, factors);
    }
}
```

`++count;` 는 원자적인 연산이 아닙니다. 실제로는 3 단계에 걸쳐 적용됩니다.

값 읽기 → 1 더하기 → 다시 쓰기



```
sequenceDiagram
    participant ThreadA
    participant ThreadB
    participant SharedCount

    Note over SharedCount: 초기값 count = 42

    ThreadA->>SharedCount: read count (42)
    ThreadB->>SharedCount: read count (42)

    ThreadA->>ThreadA: increment (42 + 1 = 43)
    ThreadB->>ThreadB: increment (42 + 1 = 43)

    ThreadA->>SharedCount: write count = 43
    ThreadB->>SharedCount: write count = 43
```

Note over SharedCount:  기대값: 44<br> 실제값: 43<br>(한 번만 증가됨)

단계	설명
두 스레드가 <b>같은 값 42</b> 를 읽음	→ 서로 영향을 모름
각각 43으로 계산	→ 둘 다 같은 결과
차례대로 43을 씀	→ <b>마지막 쓴 놈이 이김</b>
그래서 증가는 실제로 <b>한 번만 적용됨</b>	→ 이게 <b>Race Condition</b>

## 2.2.1 Race condition 경쟁 조건

경쟁 조건이란 타이밍이 딱 맞았을 때만 정답을 얻는 경우..

그 결과는 타이밍에 따라 바뀌며,

이건 우리가 원하는 **일관성(consistency)**과 **예측 가능성(predictability)**을 해칩니다. 위의 코드가 대표적인 check then act의 경쟁 조건 예시

## 2.2.2 Lazy initialization 에서의 경쟁 조건

**Lazy initialization** → 객체가 필요할 때 로딩하는 것 jpa의 지연로딩과 비슷한 개념 ㅎ

```
//잘못된 예시.  
@NotThreadSafe  
public class LazyInitRace {  
    private ExpensiveObject instance = null;  
  
    public ExpensiveObject getInstance() {  
        if (instance == null)  
            instance = new ExpensiveObject();  
        return instance;  
    }  
}
```

예를 들어, 스레드 A와 B가

동시에 `getInstance()` 를 호출한다고 가정합니다.

A는

```
instance == null
```

임을 보고

```
ExpensiveObject
```

를 생성합니다. B도 마찬가지로

```
instance == null
```

을 확인합니다.

이 시점에서 B가 `instance`를 검사했을 때의 결과는 스케줄링, A의 객체 생성 시간 등 **타이밍에 따라 달라집니다**. 만약 B가 검사할 때 여전히 `null`이라면, 두 스레드 모두 서로 **다른 객체**를 생성하게 되버립니다. ^^.

**결국 `getInstance()`는 항상 같은 인스턴스를 반환해야 한다는 계약을 깨트리게 됩니다.**

## 2.2.3 복합 동작

경쟁 조건(race condition)을 피하려면,

우리가 어떤 변수의 값을 **수정하는 중간**에는 다른 스레드가 그 변수에 접근하지 못하게 막을 수 있는 방법이 있어야 한다.

그래야 다른 스레드가 그 변수의 상태를 관찰하거나 변경하더라도,

그건 우리가 **수정을 시작하기 전**이거나, **끝난 후** 일 수는 있어도, **도중에는 안됩니다**.

“다 끝났거나, 시작도 안 했거나” — 중간은 없음 → 이게 진정한 원자성

**스레드 안전성을 보장하기 위해** `check-then-act` 연산 (예: lazy initialization)이나 `read-modify-write` 연산 (예: increment)은 **항상 원자적으로 실행되어야** 합니다.

## 아토믹 클래스 들여다보기

```
@ThreadSafe
public class CountingFactorizer implements Servlet {
    //원자성을 보장하는 Atomic 클래스를 사용하기 ㅎㅎ
    //상태를 사용했지만 안전하게 되었쥬?
    private final AtomicLong count = new AtomicLong(0);

    public long getCount() {
        return count.get();
    }

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        count.incrementAndGet();
        encodeIntoResponse(resp, factors);
    }
}
```

## 2.3 락

그렇다면 서블릿과 같은 멀티스레드 환경에서 더 많은 상태를 사용하고 싶으면 **Atomic** 객체를 띄워볼까요?

### 문제 상황: 캐싱

우리는 최근 계산한 결과를 캐싱해서 다음 클라이언트가 같은 숫자를 요청했을 때 빠르게 응답하고자 한다. 이를 위해서 두 가지 정보를 기억해야 한다:

- 마지막으로 인수분해한 숫자
- 그 숫자의 인수분해 결과

```
// Listing 2.5 - 스레드 안전하지 않은 캐시 서블릿
@NotThreadSafe
public class UnsafeCachingFactorizer implements Servlet {
```

```

private final AtomicReference<BigInteger> lastNumber
    = new AtomicReference<BigInteger>();
private final AtomicReference<BigInteger[]> lastFactors
    = new AtomicReference<BigInteger[]>();

public void service(ServletRequest req, ServletResponse resp) {
    BigInteger i = extractFromRequest(req);
    if (i.equals(lastNumber.get()))
        encodeIntoResponse(resp, lastFactors.get());
    else {
        BigInteger[] factors = factor(i);
        lastNumber.set(i);
        lastFactors.set(factors);
        encodeIntoResponse(resp, factors);
    }
}
}

```

위 코드는 두 값( `lastNumber`, `lastFactors` ) 각각을 `AtomicReference` 로 관리하고 있어 각각은 원자적이지만, 둘 사이의 관계를 보장하지는 못합니다..

즉, 하나는 업데이트됐지만 다른 하나는 업데이트되지 않은 상태가 될 수 있다 → **일관성 깨짐!**

### 2.3.1 암묵적인 락

이 문제를 해결하려면 두 값을 하나의 원자적 단위로 업데이트해야 합니다. Java에서는 이를 위해 `synchronized` 키워드를 사용한 **\*\*내장 락(intrinsic lock)\*\***이 제공됩니다.

```

// Listing 2.6 - 동기화는 했지만 성능이 나쁜 예
@ThreadSafe
public class SynchronizedFactorizer implements Servlet {
    @GuardedBy("this") private BigInteger lastNumber;
    @GuardedBy("this") private BigInteger[] lastFactors;

    public synchronized void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);

```

```

        if (i.equals(lastNumber))
            encodeIntoResponse(resp, lastFactors);
        else {
            BigInteger[] factors = factor(i);
            lastNumber = i;
            lastFactors = factors;
            encodeIntoResponse(resp, factors);
        }
    }
}

```

여기서는 `synchronized` 메서드를 사용해 **서블릿 객체(this)**를 락으로 사용

했고, 두 상태 변수는 이 락에 의해 보호받고 있습니다.

하지만 **단점**은? → 모든 요청이 메서드가 끝날때까지 기다려야 되므로 성능이 떨어질 수 있음!()

## 2.3.2 재 진입성

java의 락은 재진입 가능합니다.

이미 락을 잡은 스레드가 같은 락을 다시 획득해도 문제가 없다.

```

// Listing 2.7 - 재진입성이 없었다면 교착 상태 발생
public class Widget {
    public synchronized void doSomething() { ... }
}

public class LoggingWidget extends Widget {
    public synchronized void doSomething() {
        System.out.println(toString() + ": calling doSomething");
        super.doSomething(); // 이 부분에서 자기 자신 락 재진입
    }
}

```

재진입성 덕분에 위와 같은

**자연스러운 상속 구조**에서도 교착 상태 없이 동작할 수 있습니다..

## 2.4 락으로 상태보호하기

락을 사용하면 공유 상태에 대한 **배타적 접근**을 보장할 수 있습니다.

- 공유 변수를 사용할 때는 항상 동일한 락을 사용해야 합니다.
- 락은 읽기/쓰기 모두에 사용되어야 합니다. → 단순히 쓸 때만 동기화 하면 안됩니다. → 왜? 가시성 문제를 일으킬 수 있기 때문입니다. (3장에서 다룬다! ㅎㅎ)

```
@GuardedBy("this") private int count;
public synchronized int getCount() {
    return count;
}
```

## 2.5 활동성과 성능

### SynchronizedFactorizer의 문제

```
public synchronized void service(...) {
    // 전체 메서드 락
}
```

- 이 구조는 **모든 요청을 직렬화**시킴.
- 결과적으로 **멀티스레딩의 장점을 못 살림**.
- 특히 CPU 코어가 많은 경우에도 **병렬 실행이 불가능**

### 해결 방법

- 락의 범위를 좁히는 것 (fine-grained locking)
- 계산 로직(factor)은 **락 없이 실행**, 상태 접근 부분만 synchronized 블록으로 보호



종종 단순성과 성능이 서로 상충할 때가 있습니다. 동기화 정책을 구현할 때는 성능을 위해서 조금하게 단순성을 희생하고픈 욕심을 버려야합니다.

복잡하고 오래 걸리는 계산 작업, 네트워크 작업, 사용자 입출력 작업과 같이 빨리 끝나지 않을 수 있는 작업을 하는 부분에서는 가능한 락을 잡지 말아야합니다.