Java & Spring 동시성

Java와 Spring에서의 동시성 이슈 해결 방법

OS-Challenge 3주 차 - 심수현

INDEX

동시성 문제

프로세스 동기화와 동시성 문제

프로세스 동기화

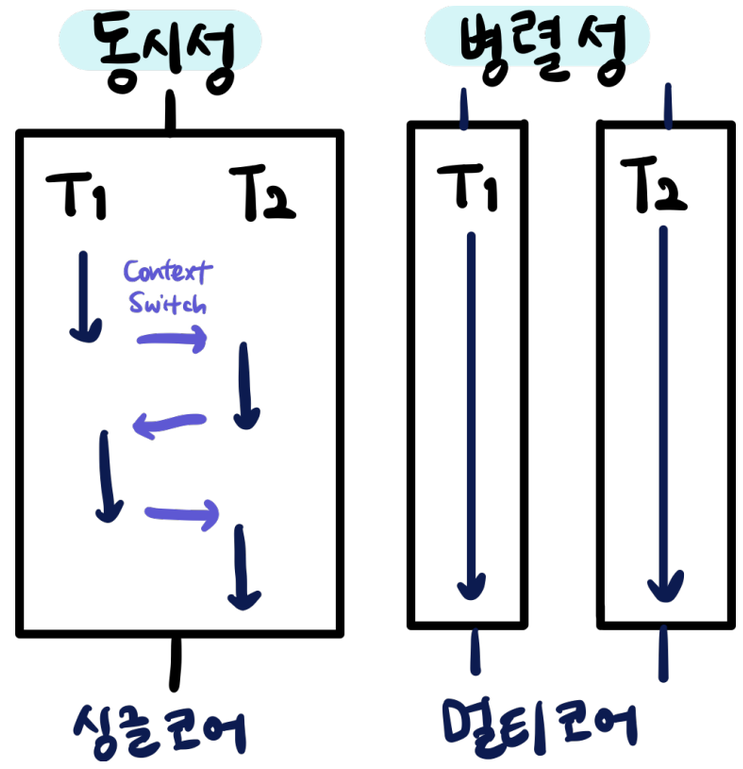
- 다중 프로세스 환경에서 자원 등에 한 프로세스만이 접근가능하도록 하는 것을 의미
- 프로세스 동기화를 하지 않으면 동시에 공유 자원에 접근할 수 있어 데이터의 일관성이 깨질 수 있음

멀티 스레드와 동기화

- 한 프로세스 안에서 실행되는 여러 개의 스레드들은 서로 공유할 수 있는 메모리 부분이 있고 이 때 동기화 문제가 발생할 수 있음
- 동기화 문제는 스레드 간의 공유자원에 접근하는 것을 한 번에 한 스레드만 가능하게 해서 해결 가능

동시성(Concurrency)과 병렬성(Parallelism)

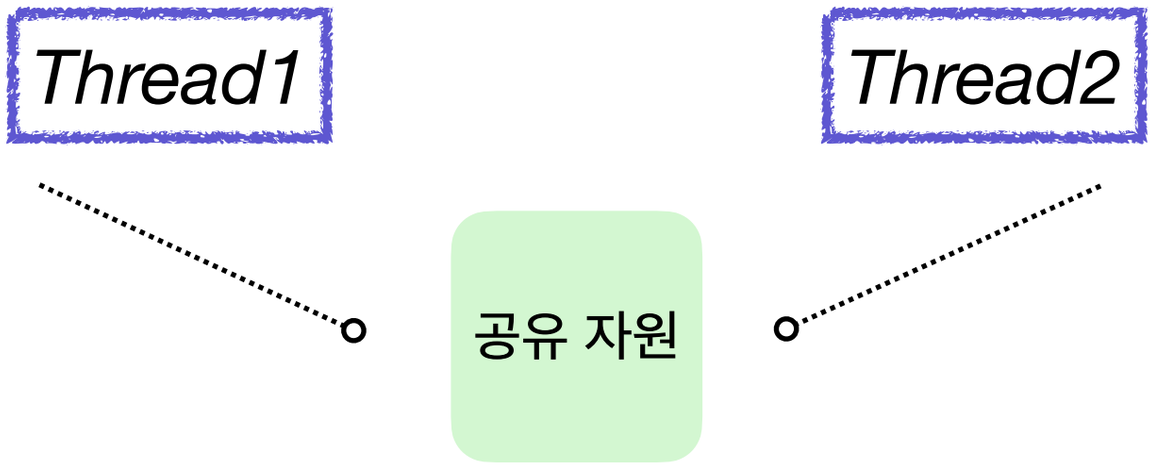
- **동시성**
  - 동시에 실행되는 것처럼 보이는 것
  - 싱글 코어에서 멀티 스레드 동작하는 방식(여러 스레드가 번갈아가며 실행)
  - 논리적 개념
  - **동시성 문제**: 동일한 자원에 대해 여러 스레드가 동시에 접근하면서 발생하는 문제
- **병렬성**
  - 실제로 여러 작업 처리를 하며 동시에 실행되는 것
  - 멀티 코어에서 멀티 스레드를 동작하는 방식으로 한 개 이상의 스레드를 포함하는 각 코어들이 동시에 실행되는 성질
  - 물리적인 개념



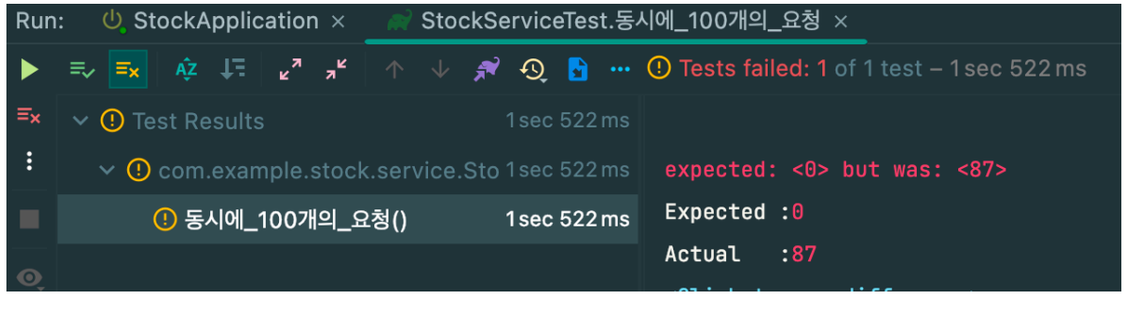
Java & Spring과 동시성

예제: 동시성 문제가 발생

```java
@Test
public void 동시에_100개의_요청() throws InterruptedException {
    // 100개의 요청
    int threadCount = 100;

    // 비동기 작업 단순화
    ExecutorService executorService = Executors.newFixedThreadPool( nThreads: 32);
    // 100개 요청 끝날 때까지 기다리기
    CountDownLatch latch = new CountDownLatch(threadCount);

    for (int i = 0; i < threadCount; i++) {
        executorService.submit(() -> {
            try {
                stockService.decrease( id: 1L,  quantity: 1L);
            } finally {
                latch.countDown();
            }
        });
    }

    latch.await();

    Stock stock = stockRepository.findById(1L).orElseThrow();

    // 처음 100개 수량 - 1개씩 100번 감소 => 0개의 수량이 남을 것으로 예상
    assertEquals( expected: 0L, stock.getQuantity());
}
```



- 값을 어디선가 변경할 때 발생(읽기말 할 때는 발생X)
- 여러 스레드가 동시에 하나의 자원을 공유하고 있기 때문에 같은 자원을 두고 경쟁 상태(Race Condition)과 같은 문제가 발생



Java & Spring과 동시성

동기화 키워드 1) synchronized - 암시적 Lock

Synchronized

- Java의 예약어(변수명이나 클래스명으로 사용 불가능)
- synchronized를 이용해서 임계구역 설정
- **암시적 락**: Lock 클래스를 직접 사용해서 lock, unlock을 구현하는 것이 아닌 내부적으로 객체의 고유 락을 사용해서 접근을 제어
- **고유 락(intrinsic lock)**: 자바의 모든 객체가 갖고 있는 락(monitor lock), synchronized는 고유 락 사용

사용 방법

1) **synchronized methods**: 메소드 자체를 synchronized로 선언

```
public synchronized void methodName() {

}
```

스레드는 synchronized 키워드가 붙은 메서드가 호출된 시점부터
해당 메서드가 포함된 객체의 lock을 얻어 작업을 수행하다가
메서드가 종료되면 lock 반환

2) **synchronized statements**: 메소드 내의 특정 문장만 synchronzied로 감싸는 방법

```
synchronzied (객체의 참조 변수) {

}
```

이 때 참조 변수는 lock을 걸고자 하는 객체를 참조하는것
이 영역으로 들어가면서부터 스레드는 지정된 객체의 lock을
얻게 되고 블록을 벗어나면 lock 반납

- 임계영역은 멀티 스레드 프로그램의 성능을 좌우 => 가능하면 메소드 전체에 lock을 걸기 보다는 synchronized 블록으로 임계구역 최소화

Synchronized의 문제점

- synchronized는 한 프로세스 내에서만 동작 => 멀티 프로세스 상황에서는?
- synchronized는 blocking을 사용하여 멀티 스레드 환경에서 공유 객체를 동기화하는 키워드 => blocking 성능 이슈 발생 가능성
  - 특정 스레드가 해당 블록 전체에 lock을 걸면 해당 lock에 접근하는 스레드들이 blocking 상태로 들어감

Java & Spring과 동시성

동기화 키워드 1) synchronized - 암시적 Lock

Synchronized

synchronized 사용 전

```
2 usages
class Account {
    3 usages
    private int balance = 1000;

    2 usages
    public int getBalance() {
        return balance;
    }

    1 usage
    public void withdraw(int money) {
        if (balance >= money) {
            try {
                Thread.sleep( millis: 1000);
            } catch (InterruptedException e) {
            }
            balance -= money;
        }
    }
}
```

```
1 usage
class MyThread1 implements Runnable {
    3 usages
    Account account = new Account();

    @Override
    public void run() {
        while (account.getBalance() > 0) {
            int money = (int) (Math.random() * 3 + 1) * 100;
            account.withdraw(money);
            System.out.println("balance: " + account.getBalance());
        }
    }
}
```

```
public class ThreadExample {
    public static void main(String[] args) {
        Runnable r = new MyThread1();
        new Thread(r).start();
        new Thread(r).start();
    }
}
```

```
> Task :ThreadExample.main()
balance: 500
balance: 700
balance: 400
balance: 300
balance: 0
balance: -200
```

Java & Spring과 동시성

동기화 키워드 1) synchronized - 암시적 Lock

Synchronized

- Synchronized 메소드

```
1 usage
public synchronized void withdraw(int money) {
    if (balance >= money) {
        try {
            Thread.sleep( millis: 1000);
        } catch (InterruptedException e) {
        }
        balance -= money;
    }
}
```

```
> Task :ThreadExample.main()
balance: 900
balance: 700
balance: 600
balance: 300
balance: 0
balance: 0
```

synchronized 사용 후

- Synchronized 블록

```
1 usage
public void withdraw(int money) {
    synchronized (this) {
        if (balance >= money) {
            try {
                Thread.sleep( millis: 1000);
            } catch (InterruptedException e) {
            }
            balance -= money;
        }
    }
}
```

```
> Task :ThreadExample.main()
balance: 900
balance: 700
balance: 500
balance: 400
balance: 300
balance: 200
balance: 0
balance: 0
```

Java & Spring과 동시성

동기화 키워드 1) synchronized - 암시적 Lock

Synchronized

wait(), notify(), notifyAll()

- 동기화된 스레드(보호구역에서 실행하고 있는 스레드) 혼자서는 동기화 블록에서 다른 스레드로 제어권을 넘길 수 없음
- 동기화된 블록에서 스레드 간의 동기화에 대한 통신을 하기 위해서는 wait(), notify(), notifyAll() 메소드를 사용
- Monitor의 Condition Variable를 통해 메소드 구현되어 있음, Object에 정의되어 있는 메소드
- ⚠️ synchronized 블록에서만 사용! (아닐 경우 java.lang.illegalMonitorStateException 발생)

- wait(): Lock을 가진 스레드가 다른 스레드에 Lock을 넘겨준 이후에 대기해야 할 때 사용
- notify(): 대기중인 임의의 스레드(우선순위가 높은 스레드)를 깨울 때 사용
- notifyAll(): 대기중인 모든 스레드를 깨울 때 사용 -> 하나의 스레드만 Lock을 획득하고 나머지 스레드는 다시 대기 상태에 들어감

Effective Java Item 81.
wait와 notify보다는 동시성 유틸리티를 애용해라

- Java 5부터 도입된 고수준의 동시성 유틸리티가 wait(), notify()로 하드코딩 해야하는 일들을 대신 처리해줌
- java.util.concurrent의 고수준 유틸리티

1) 실행자 프레임워크

```
// 작업 큐 생성
ExecutorService exec = Executors.newSingleThreadExecutor();

// 실행할 태스크 넘기기
exec.execute(runnable);

// 실행자 종료
exec.shutdown();
```

2) 동시성 컬렉션(concurrent collection)

- List, Queue, Map 같은 표준 컬렉션 인터페이스에 동시성을 구현한 고성능 컬렉션
- 높은 동시성에 도달하기 위해 동기화를 각자의 내부에서 수행
- Ex. ConcurrentHashMap, BlockingQueue,

3) 동기화 장치(synchronizer)

- 스레드가 다른 스레드를 기다릴 수 있게 하여 서로 작업을 조율할 수 있게 함
- Ex. CountDownLatch, Semaphore, Phaser

Java & Spring과 동시성

동기화 키워드 2) java.util.concurrent.lock - 명시적 Lock

명시적 *Lock*

ReentrantLock, ReentrantReadWriteLock, StampedLock

- **ReentrantLock**: 재진입이 가능한 Lock, 가장 일반적(재진입: 특정 조건에서 락을 풀었다가 나중에 다시 락을 걸 수 있음)
- **ReentrantReadWriteLock**: 읽기 & 쓰기를 위한 Lock을 제공, static 클래스로 구현
  - 무조건 lock이 있어야만 임계 영역의 코드를 수행할 수 있는 ReentrantLock과 달리,
    읽기 락이 걸려 있으면, 다른 쓰레드가 읽기 락을 중복해서 걸고 읽기를 수행할 수 있음
  - 그러나 읽기 락이 걸린 상태에서 쓰기 락을 거는 것은 허용되지 않음 (반대도 마찬가지)
- **StampedLock**: 락을 걸거나 해지할 때 '스탬프(long 타입의 정수 값)'를 사용
  - ReentrantReadWriteLock에 '낙관적 읽기 락(optimistic reading lock)'이 추가된 형태
  - 읽기 락이 걸려있으면 쓰기 락을 얻기 위해서는 읽기 락이 풀릴 때까지 기다려야 하는데 비해 낙관적 읽기 락은 쓰기 락에 의해 바로 풀림

ReentrantLock

```java
public class ReentrantLockSingleton {

    1 usage
    private int count = 0;
    2 usages
    private Lock lock = new ReentrantLock();

    public int increase() {
        // lock 획득
        lock.lock();

        try {
            return ++count;
        } finally {
            // lock 반납
            lock.unlock();
        }
    }
}
```

ReentrantReadWriteLock

```java
public class ReentrantReadWriteLockSingleton {
    3 usages
    private int count = 0;
    2 usages
    private ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
    4 usages
    private Lock readLock = lock.readLock(); // 읽기 락
    2 usages
    private Lock writeLock = lock.writeLock(); // 쓰기 락
```

```java
// 읽기 락
public int getCount() {
    readLock.lock();
    try {
        return count;
    } finally {
        readLock.unlock();
    }
}
```

```java
// 쓰기 락
public void setCount(int count) {
    readLock.lock();
    try {
        this.count = count;
    } finally {
        readLock.unlock();
    }
}
```

```java
public int increase() {
    writeLock.lock();
    try {
        return ++count;
    } finally {
        // 락 반납
        writeLock.unlock();
    }
}
```

Java & Spring과 동시성

동기화 키워드 2) java.util.concurrent.lock - 명시적 Lock

명시적 *Lock*

ReentrantLock, ReentrantReadWriteLock, StampedLock

- **ReentrantLock**: 재진입이 가능한 Lock, 가장 일반적(재진입: 특정 조건에서 락을 풀었다가 나중에 다시 락을 걸 수 있음)
- **ReentrantReadWriteLock**: 읽기 & 쓰기를 위한 Lock을 제공, static 클래스로 구현
  - 무조건 lock이 있어야만 임계 영역의 코드를 수행할 수 있는 ReentrantLock과 달리,
    읽기 락이 걸려 있으면, 다른 쓰레드가 읽기 락을 중복해서 걸고 읽기를 수행할 수 있음
  - 그러나 읽기 락이 걸린 상태에서 쓰기 락을 거는 것은 허용되지 않음 (반대도 마찬가지)
- **StampedLock**: 락을 걸거나 해지할 때 '스탬프(long 타입의 정수 값)'를 사용
  - ReentrantReadWriteLock에 '낙관적 읽기 락(optimistic reading lock)'이 추가된 형태
  - 읽기 락이 걸려있으면 쓰기 락을 얻기 위해서는 읽기 락이 풀릴 때까지 기다려야 하는데 비해 낙관적 읽기 락은 쓰기 락에 의해 바로 풀림

StampedLock

```
4 usages
StampedLock lock = new StampedLock();

int getBalance() {

    // 낙관적 읽기 락
    long stamp = lock.tryOptimisticRead();

    int currentBalance = this.balance;

    if (!lock.validate(stamp)) {    // 쓰기 락에 의해 낙관적 읽기 락이 풀렸는지 확인
        stamp = lock.readLock();    // 락이 풀렸으면, 읽기 락을 얻으려고 기다림

        try {
            currentBalance = this.balance;
        } finally {
            lock.unlockRead(stamp);           // 읽기 락 풀기
        }
    }
    return currentBalance;    // 낙관적 읽기 락이 풀리지 않았으면 곧바로 읽어온 값을 반환
}
```

Java & Spring과 동시성

동기화 키워드 2) java.util.concurrent.lock - 명시적 Lock

명시적 *Lock*

ReentrantLock, ReentrantReadWriteLock, StampedLock

- **ReentrantLock**: 재진입이 가능한 Lock, 가장 일반적(재진입: 특정 조건에서 락을 풀었다가 나중에 다시 락을 걸 수 있음)
- **ReentrantReadWriteLock**: 읽기 & 쓰기를 위한 Lock을 제공, static 클래스로 구현
  - 무조건 lock이 있어야만 임계 영역의 코드를 수행할 수 있는 ReentrantLock과 달리,
    읽기 락이 걸려 있으면, 다른 쓰레드가 읽기 락을 중복해서 걸고 읽기를 수행할 수 있음
  - 그러나 읽기 락이 걸린 상태에서 쓰기 락을 거는 것은 허용되지 않음 (반대도 마찬가지)
- **StampedLock**: 락을 걸거나 해지할 때 '스탬프(long 타입의 정수 값)'를 사용
  - ReentrantReadWriteLock에 '낙관적 읽기 락(optimistic reading lock)'이 추가된 형태
  - 읽기 락이 걸려있으면 쓰기 락을 얻기 위해서는 읽기 락이 풀릴 때까지 기다려야 하는데 비해 낙관적 읽기 락은 쓰기 락에 의해 바로 풀림

Cf.) Condition

- synchronized의 wait(), notify()는 스레드의 종류를 구분하지 않고 공유 객체의 waiting pool에 같이 몰아 넣어 선별적인 통지가 불가능 했지만,
- ReentrantLock과 Condition을 사용하면 쓰레드의 종류에 따라 구분된 waiting pool에서 따로 기다리도록 하여 선별적인 통지를 가능하게 함
- Condition을 이용하면 각각의 waiting pool에서 쓰레드의 종류에 따라 구분하여 넣을 수 있음
- => 경쟁 상태가 발생할 가능성이 낮아짐

Object	Condition
void wait()	void await() void awaitUninterruptibly()
void wait(long timeout)	boolean await(long time, TimeUnit unit) long awaitNanos(long nanosTimeout) boolean awaitUntil(Date deadline)
void notify()	void signal()
void notifyAll()	void signalAll()

Java & Spring과 동시성

동기화 키워드 3) Atomic Type - concurrent package 사용

Atomic Type

- Atomicity(원자성): 쪼갤 수 없는 가장 작은 단위
- Java의 Atomic Type은 Wrapping 클래스의 일종으로, 참조 타입과 원시 타입 두 종류의 변수에 모두 적용이 가능
- java.util.concurrent.atomic 패키지에 정의된 클래스
- 사용시 내부적으로 CAS(Compare-And-Swap) 알고리즘을 사용해 lock 없이 동기화 처리를 할 수 있음
  - CAS는 특정 메모리 위치와 주어진 위치의 value를 비교하여 다르면 대체하지 않음

사용 방법

- 변수를 선언할 때 타입을 Atomic Type으로 선언
- 주요 클래스
  - AtomicBoolean / AtomicLong / AtomicIntegerArray / AtomicDoubleArray
- 주요 메소드
  - get()
  - set(newValue): 값 업데이트
  - getAndSet(newValue): 원자적으로 값 업데이트 후 원래 값 반환
  - compareAndSet(expect, update): 현재값과 예상값이 동일한 경우 update 후 true 반환, 그렇지 않을 경우 update없이 false만 반환

Compare-And-Swap(CAS)

- 메모리 위치의 내용을 주어진 값과 비교 후 동일한 경우에만 해당 메모리 위치의 내용을 새로 주어진 값으로 수정
- 현재 주어진 값(현재 스레드에서의 데이터)과 실제 데이터를 비교해서 두 개가 일치할 때만 값을 업데이트 => ***compareAndSet()***
- 현재 연산 주에서 스레드의 값과 메모리의 값이 다른 경우 중간에 다른 스레드를 토한 작업이 있었던 것으로 판단하여 *write* 중단하고 작업 재시도

Java & Spring과 동시성

동기화 키워드 4) volatile

Volatile

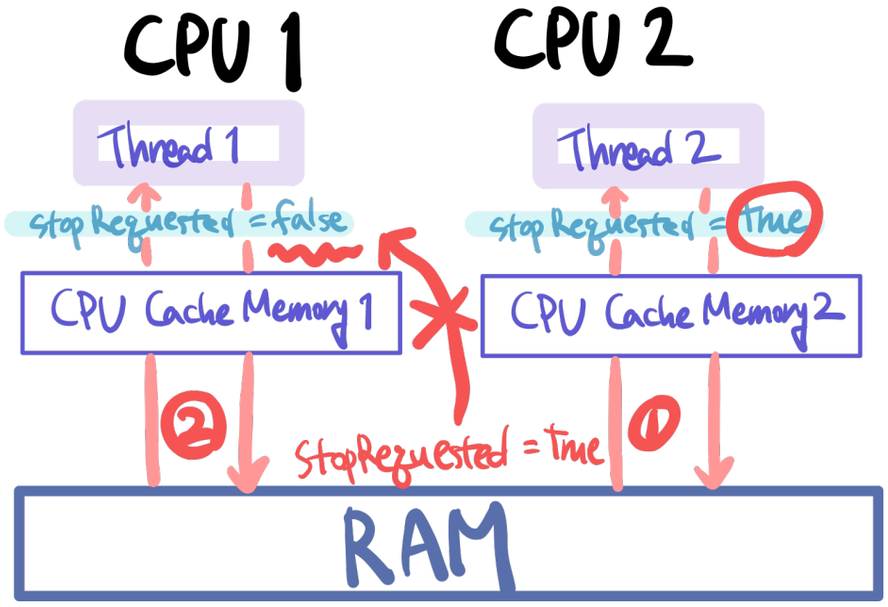
- 동시성 프로그래밍에서 발생할 수 있는 문제 중 하나인 **가시성 문제**를 해결하기 위해 사용되는 키워드
- volatile 키워드는 Java 변수를 Main Memory에 저장하겠다라는 것을 명시
- 매번 변수의 값을 Read할 때마다 CPU cache에 저장된 값이 아닌 Main Memory에서 읽는 것
- 또한 변수의 값을 Write할 때마다 Main Memory에 작성

🤔 **Volatile 가시성?**

- Volatile 변수를 사용하고 있지 않는 MultiThread 애플리케이션은 작업을 수행하는 동안 성능 향상을 위해서 Main Memory에서 읽은 변수를 **CPU Cache에 저장**
- 만약 Multi Thread환경에서 Thread가 변수 값을 읽어올 때 각각의 CPU Cache에 저장된 값이 다르기 때문에 변수 값 불일치 문제가 발생
- => ✅ 여러 개의 스레드가 사용됨에 따라, CPU Cache Memory와 RAM의 데이터가 서로 일치하지 않아 생기는 문제
- volatile 키워드를 붙인 공유 자원은 RAM에 직접 읽고 쓰는 작업을 수행할 수 있도록 해줌

```java
public class Volatile {

    2 usages
    private static boolean stopRequested;

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

        Thread.sleep( millis: 1000);
        stopRequested = true;
    }
}
```

```
stock [:Vo 48 sec
:Volatile. 46 sec

5:03:14 PM: Executing ':Volatile.main()'...

> Task :compileJava
> Task :processResources UP-TO-DATE
> Task :classes
> Task :Volatile.main()
```



Java & Spring과 동시성

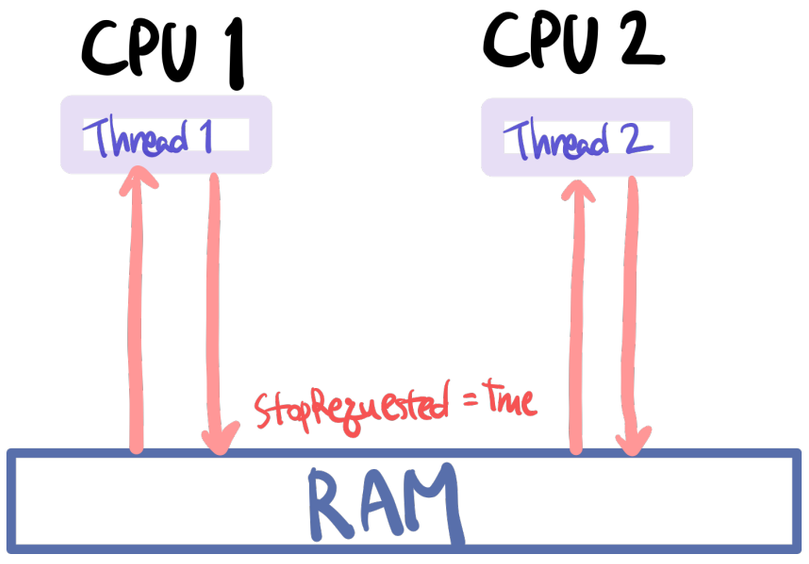
동기화 키워드 4) volatile

Volatile

```
2 usages
private static volatile boolean stopRequested;

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

    Thread.sleep( millis: 1000);
    stopRequested = true;
}
```

stoc 2 sec, 459 ms

```
5:17:11 PM: Executing ':Volatile.main()'...

> Task :compileJava
> Task :processResources UP-TO-DATE
> Task :classes
> Task :Volatile.main()
```



🤔 Volatile은 어떤 상황에 사용할까요?

- Multi Thread의 안정성(데이터 무결성)을 확보한다고 여기저기 synchronized 혹은 lock을 남발한다면 Multi Thread 로직으로 인해 코드 복잡도만 높아지고, 실제 성능에 대한 효과는 크게 누리지 못할 것이기에 어떤 형태던 lock을 최소화하는 것이 좋은데 volatile도 그 방법 중 하나
- ⚠️ 단, 하나의 Thread만이 연산(modify)을 해야 함, 만약 이 전제가 확실한 경우라면
  변수에 volatile로만 lock 없이도 문제 없는 데이터를 사용할 수 있음

Java & Spring과 동시성

동기화 키워드 4) volatile

Volatile

❓ **가시성이 보장되면 동시성도 보장될까? => (X)**

- Volatile 키워드는 volatile 변수를 메인 메모리로부터 읽을 수 있게 해주는 것이 전부이며 다른 스레드에 의해 이 값이 언제든지 바뀔 수 있음
- 즉, 가시성이란 공유 데이터를 읽는 경우의 동시성만 보장해줌
- 가시성 vs 원자성
  - 가시성: CPU - Cache - Memory 관계상의 개념 (메모리 가시성)
  - 원자성: 한 줄의 프로그램 문장이 컴파일러에 의해 기계어로 변경되면서, 이를 기계가 순차적으로 처리하기 위한 여러 개의 Machine Instruction이 만들어져 실행되기 때문에 일어나는 현상 (연산의 원자성)

Ex. Q. 0이 들어있는 변수(i)에 10개의 스레드가 동시에 접근해서 `i++` 연산을 하면 우리 예상과 다르게 10이 나오지 않습니다. 왜 그럴까요?

📌 **++ 연산의 구체적인 동작:**

- i의 기존 값을 읽는다. (READ)
- i에 1을 더한다. (MODIFY)
- i의 값을 변수에 할당한다. (WRITE)

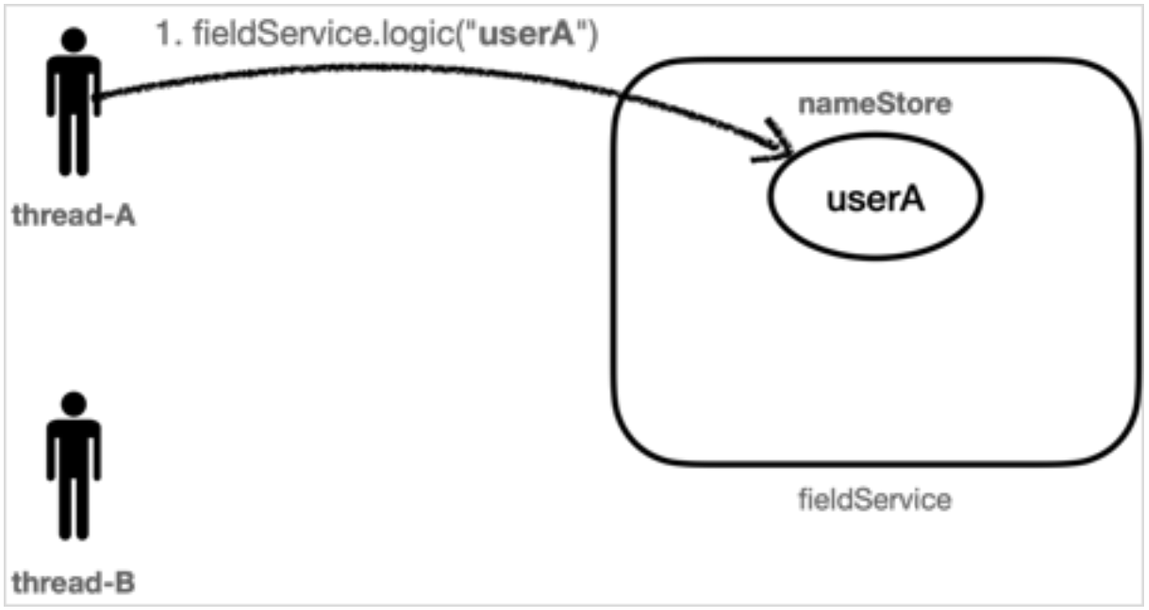
🤔 이를 두 개 Thread가 동시에 100회 수행한다고 했을때, 만약 i++이 **원자성**을 가지고 있는 연산이라고 하면
=> 결과 200이어야 하겠지만, 실제로는 200보다 작은 값이 도출됨

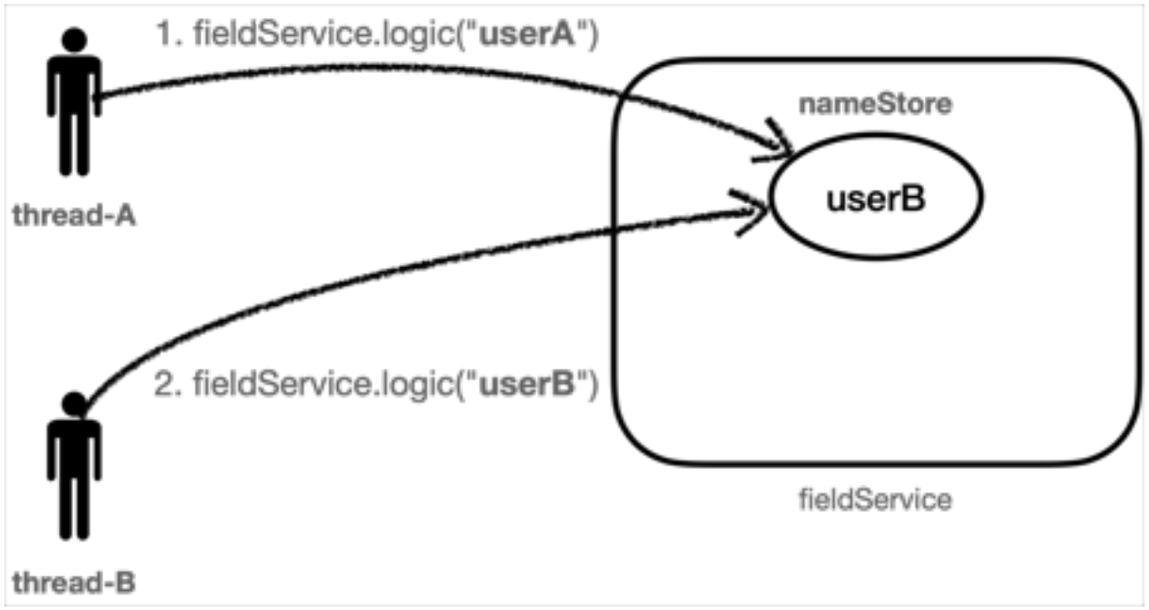
✅ 원인: i++이 3개의 instruction(READ-MODITY-WRITE)로 구성되어있기 때문!
Thread1이 값을 읽어 i+1을 하기 직전에 Thread2가 i를 읽어 i+1을 수행하고 반영하는 동작을 수행한다면 후자의 연산은 무효가 되는 현상이 발생함
즉, 가시성 문제를 해결하더라도 원자성이 확보되지 못하면 원치 않는 결과가 도출될 수 있음

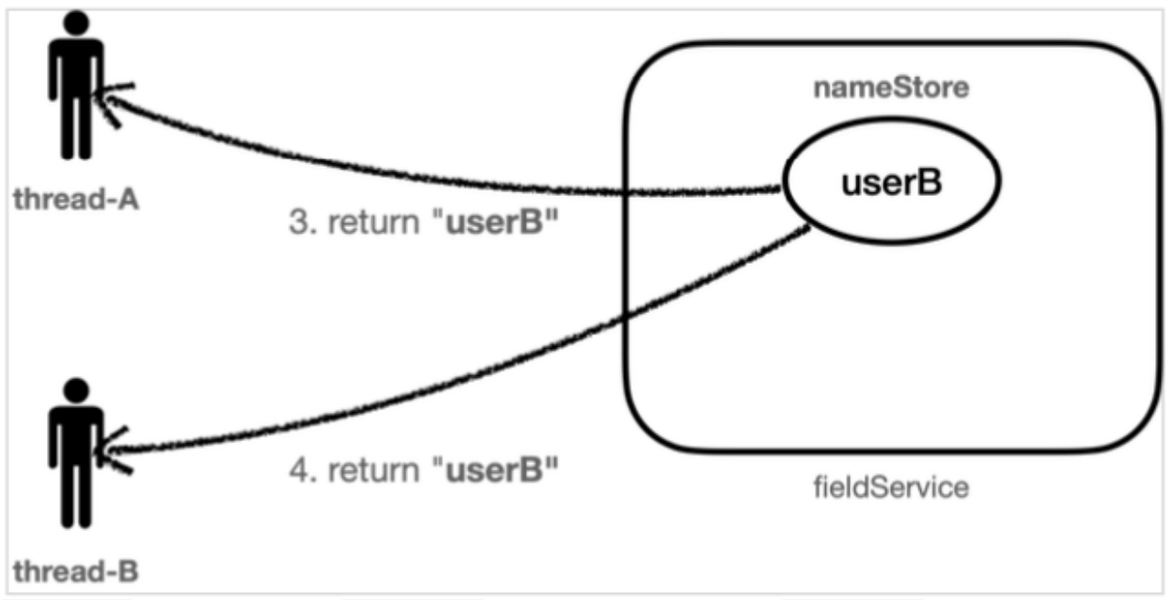
Java & Spring과 동시성

동기화 키워드 5) ThreadLocal

Thread Local







1. Thread-A 는 userA 를 nameStore 에 저장
2. Thread-B 는 userB 를 nameStore 에 저장
3. Thread-A 는 userB 를 nameStore 에서 조회
4. Thread-B 는 userB 를 nameStore 에서 조회

=> 동시성 문제: Thread-A 입장에서는 저장한 데이터와 조회한 데이터가 다른 문제가 발생

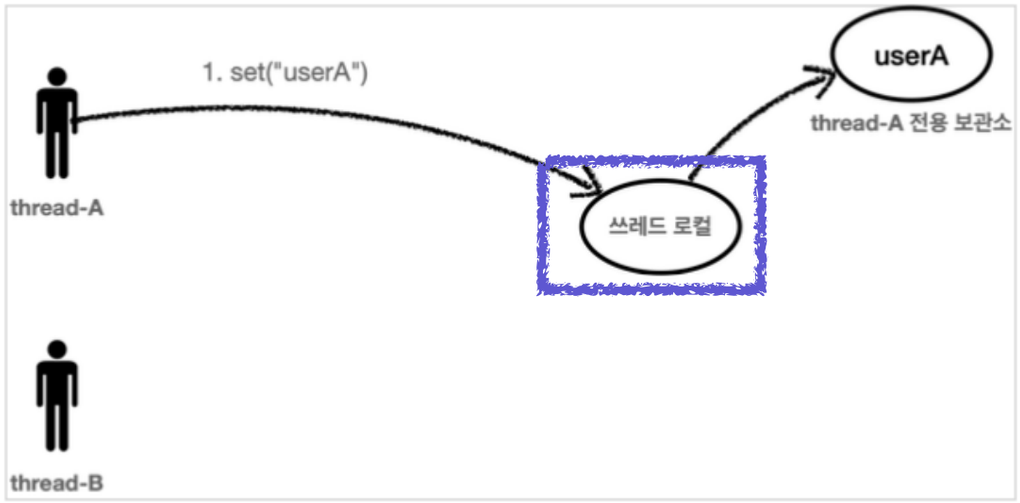
- 여러 쓰레드가 동시에 같은 인스턴스의 필드 값을 변경하면서 발생하는 문제
- 여러 쓰레드가 같은 인스턴스의 필드에 접근해야 하기 때문에 트래픽이 적은 상황에서는 확률상 잘 나타나지 않고, **트래픽이 점점 많아질 수록** 자주 발생
- 특히 스프링 빈 처럼 **싱글톤 객체의 필드를 변경**하며 사용할 때 주의

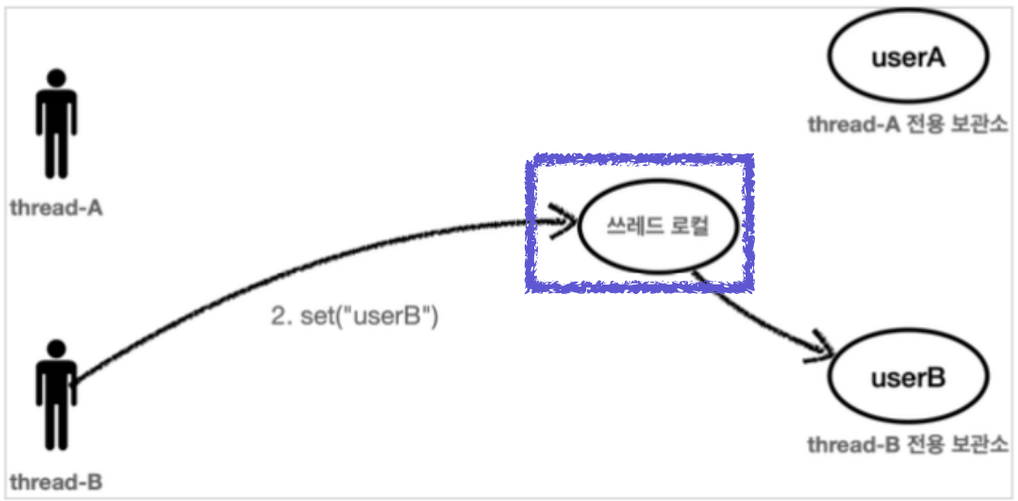
Java & Spring과 동시성

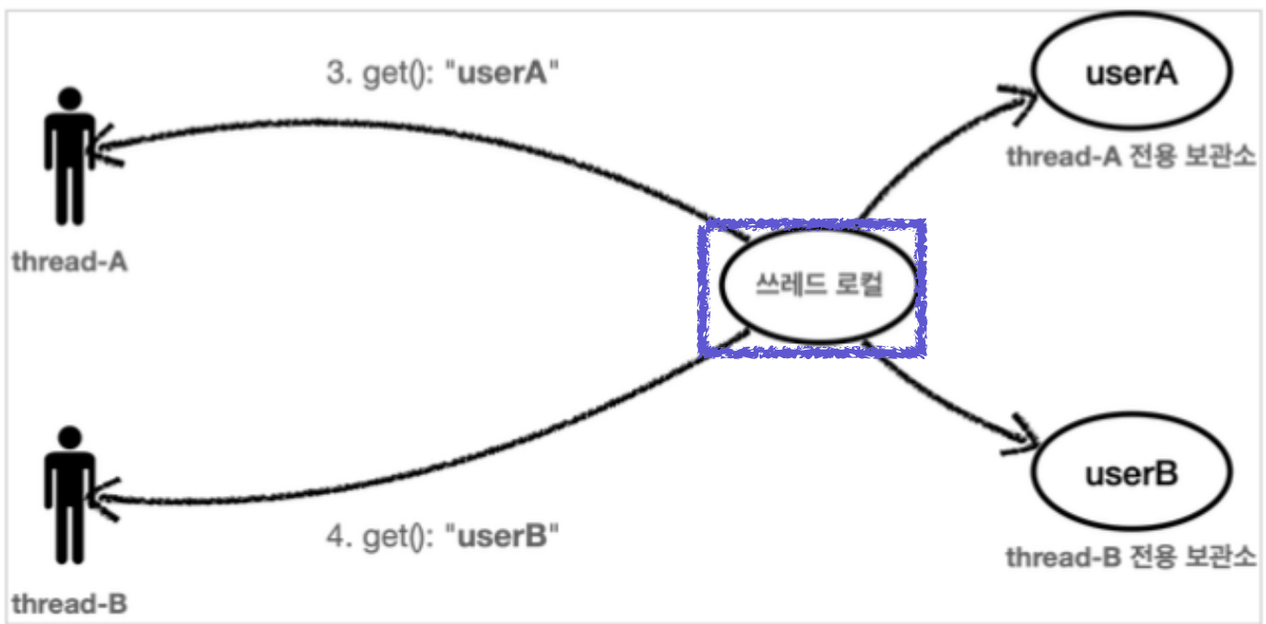
동기화 키워드 5) ThreadLocal

Thread Local

- 쓰레드 로컬은 해당 쓰레드만 접근할 수 있는 특별한 저장소 (ex. 물건 보관 창구)
  - 여러 사람이 같은 물건 보관 창구를 사용하더라도 창구 직원은 사용자를 인식해서 사용자별로 확실하게 물건을 구분
  - 사용자A, 사용자B 모두 창구 직원을 통해서 물건을 보관하고, 꺼내지만 창구 지원이 사용자에 따라 보관한 물건을 구분해주는 것







- 쓰레드 로컬을 사용하면 각 쓰레드마다 별도의 내부 저장소를 제공
  => 따라서 같은 인스턴스의 쓰레드 로컬 필드에 접근해도 문제 X
- 쓰레드 로컬을 통해서 데이터를 조회할 때
  - thread-A 가 조회하면 쓰레드 로컬은
    thread-A 전용 보관소에서 userA 데이터를 반환
  - thread-B 가 조회하면 thread-B 전용 보관소에서
    userB 데이터를 반환
- 자바는 언어차원에서 쓰레드 로컬을 지원하기 위한
  java.lang.ThreadLocal 클래스를 제공

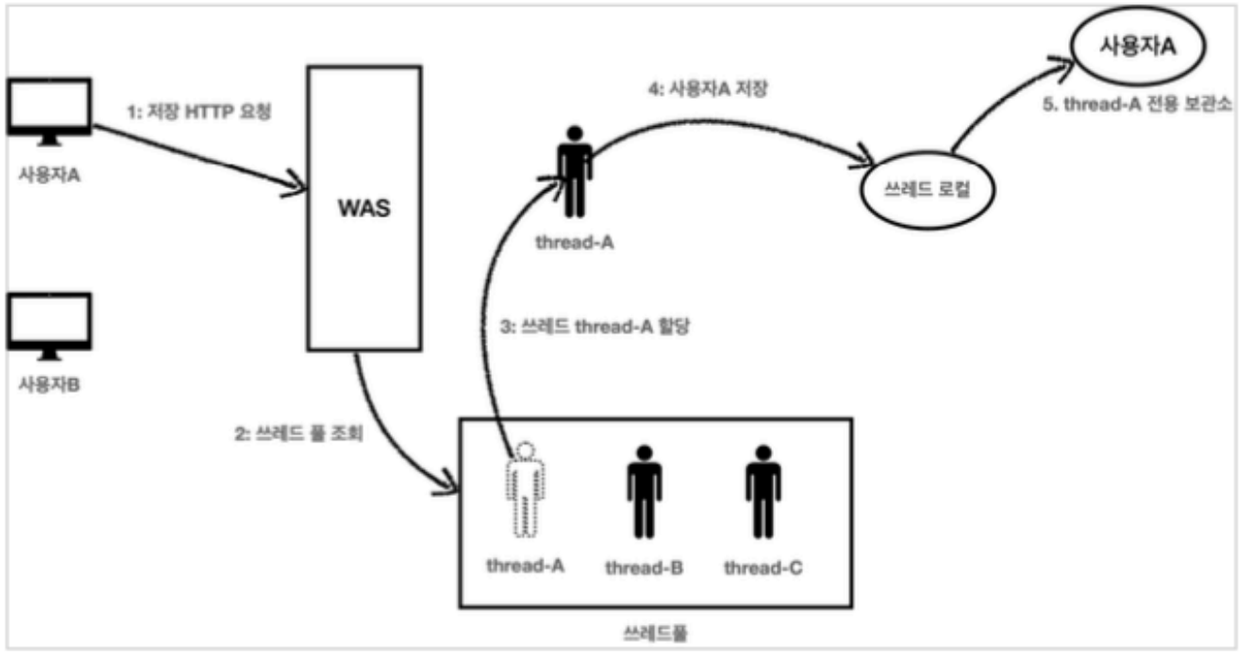
Java & Spring과 동시성

동기화 키워드 5) ThreadLocal

Thread Local

사용 시 주의점

- 쓰레드 로컬의 값을 사용 후 제거하지 않고 그냥 두면
  WAS(톰캣)처럼 쓰레드 풀을 사용하는 경우에 심각한 문제가 발생



사용자 A 저장 요청

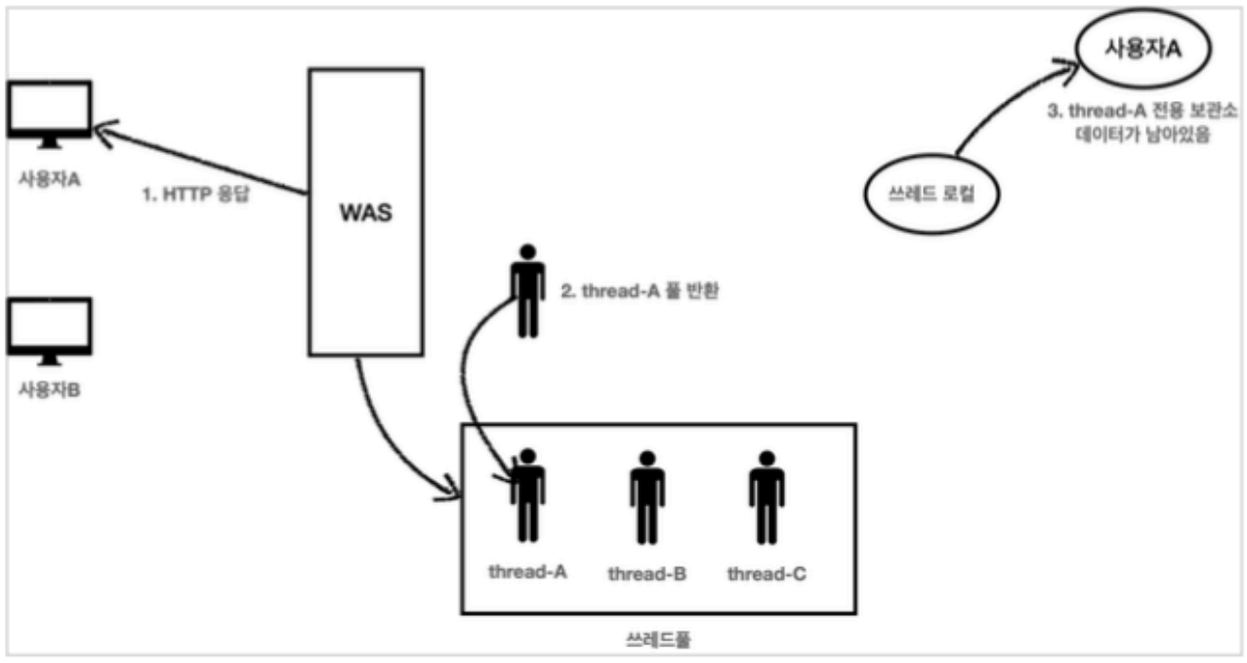
. 사용자A가 저장 HTTP를 요청했다.

. WAS는 쓰레드 풀에서 쓰레드를 하나 조회한다.

. 쓰레드 `thread-A` 가 할당되었다.

. `thread-A` 는 `사용자A` 의 데이터를 쓰레드 로컬에 저장한다.

. 쓰레드 로컬의 `thread-A` 전용 보관소에 `사용자A` 데이터를 보관한다.



사용자 A 저장 요청 종료

1. 사용자A의 HTTP 응답이 끝난다.
2. WAS는 사용이 끝난 `thread-A` 를 쓰레드 풀에 반환한다. 쓰레드를 생성하는 비용은 비싸기 때문에 쓰레드를 제거하지 않고, 보통 쓰레드 풀을 통해서 쓰레드를 재사용한다.
3. `thread-A` 는 쓰레드풀에 아직 살아있다. 따라서 쓰레드 로컬의 `thread-A` 전용 보관소에 `사용자A` 데이터도 함께 살아있게 된다.

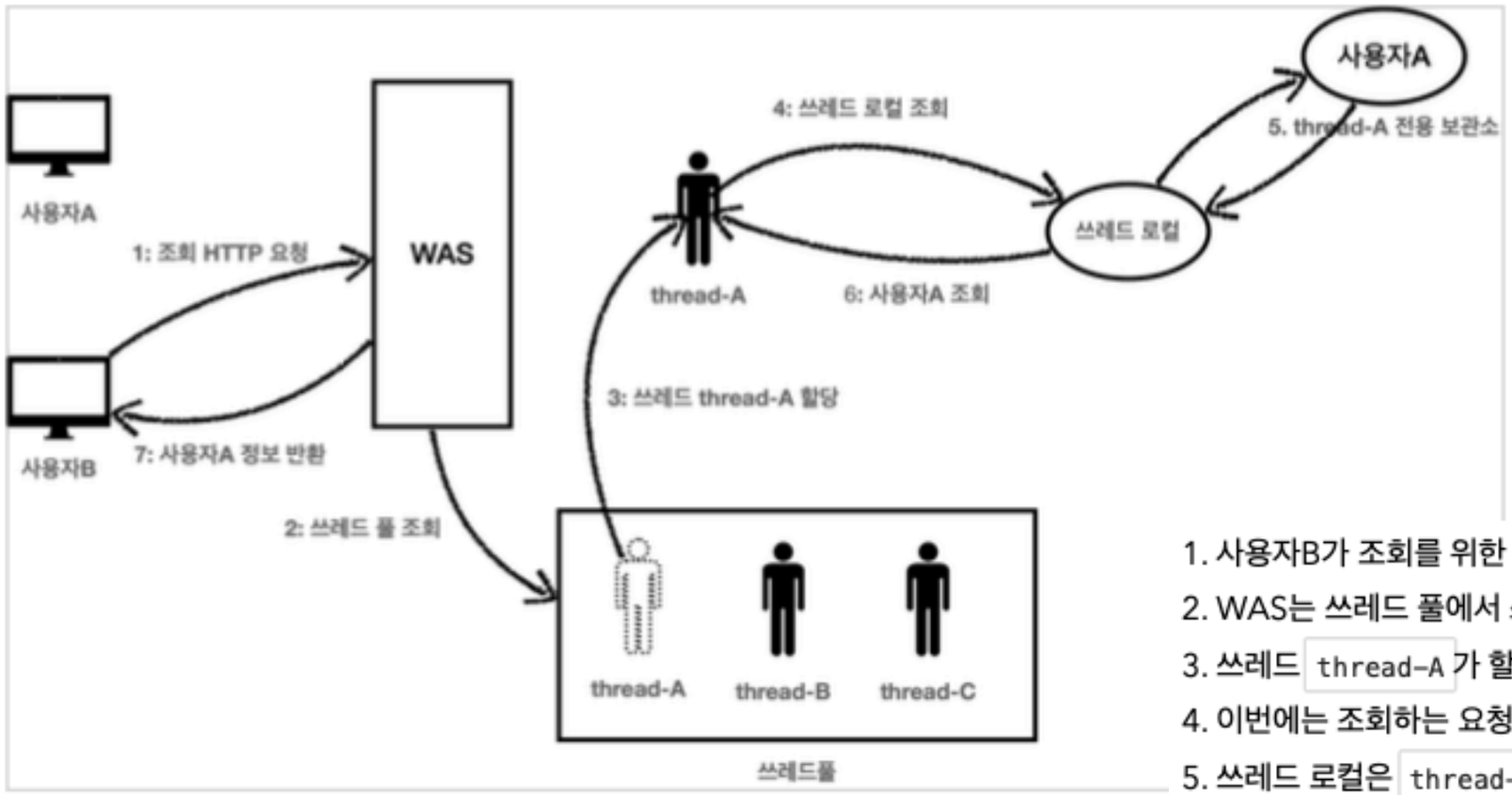
Java & Spring과 동시성

동기화 키워드 5) ThreadLocal

Thread Local

사용 시 주의점

- 쓰레드 로컬의 값을 사용 후 제거하지 않고 그냥 두면
  WAS(톰캣)처럼 쓰레드 풀을 사용하는 경우에 심각한 문제가 발생



1. 사용자B가 조회를 위한 새로운 HTTP 요청을 한다.
2. WAS는 쓰레드 풀에서 쓰레드를 하나 조회한다.
3. 쓰레드 `thread-A`가 할당되었다. (물론 다른 쓰레드가 할당될 수 도 있다.)
4. 이번에는 조회하는 요청이다. `thread-A`는 쓰레드 로컬에서 데이터를 조회한다.
5. 쓰레드 로컬은 `thread-A` 전용 보관소에 있는 `사용자A` 값을 반환한다.
6. 결과적으로 `사용자A` 값이 반환된다.
7. 사용자B는 사용자A의 정보를 조회하게 된다.

✅ **ThreadLocal.remove()를 통해 스레드 로컬의 값을 꼭 제거하기**

Quiz

- Q1. 0이 들어있는 변수에 10개의 스레드가 동시에 접근해서 ++ 연산을 하면 우리 예상과 다르게 10이 나오지 않습니다. 왜 그럴까요?
- Q2. Java에서 동시성과 관련된 예약어를 모두 말씀해주세요. (가시성, 원자성의 개념을 포함해서!)
- Q3. ThreadLocal을 사용할 때의 주의점은 무엇일까요?