2. 쓰레드 로컬 - ThreadLocal

#인강/6.핵심 원리 - 고급편/강의#

목차

필드 동기화 - 개발

앞서 로그 추적기를 만들면서 다음 로그를 출력할 때 `트랜잭션ID`와 `level`을 동기화 하는 문제가 있었다.
이 문제를 해결하기 위해 `TraceId`를 파라미터로 넘기도록 구현했다.
이렇게 해서 동기화는 성공했지만, 로그를 출력하는 모든 메서드에 `TraceId` 파라미터를 추가해야 하는 문제가 발생했다.
`TraceId`를 파라미터로 넘기지 않고 이 문제를 해결할 수 있는 방법은 없을까?

이런 문제를 해결할 목적으로 새로운 로그 추적기를 만들어보자.
이제 프로토타입 버전이 아닌 정식 버전으로 제대로 개발해보자.
향후 다양한 구현제로 변경할 수 있도록 `LogTrace` 인터페이스를 먼저 만들고, 구현해보자.

LogTrace 인터페이스

```
package hello.advanced.trace.logtrace;

import hello.advanced.trace.TraceStatus;

public interface LogTrace {
```

```
    TraceStatus begin(String message);
    void end(TraceStatus status);
    void exception(TraceStatus status, Exception e);
}
```

`LogTrace` 인터페이스에는 로그 추적기를 위한 최소한의 기능인 `begin()`, `end()`, `exception()`를 정의했다.
이제 파라미터를 넘기지 않고 `TraceId`를 동기화 할 수 있는 `FieldLogTrace` 구현체를 만들어보자.

FieldLogTrace

```
package hello.advanced.trace.logtrace;

import hello.advanced.trace.TraceId;
import hello.advanced.trace.TraceStatus;
import lombok.extern.slf4j.Slf4j;

@Slf4j
public class FieldLogTrace implements LogTrace {

    private static final String START_PREFIX = "-->";
    private static final String COMPLETE_PREFIX = "<--";
    private static final String EX_PREFIX = "<X-";

    private TraceId traceIdHolder; //traceId 동기화, 동시성 이슈 발생

    @Override
    public TraceStatus begin(String message) {
        syncTraceId();
        TraceId traceId = traceIdHolder;
        Long startTimeMs = System.currentTimeMillis();
        log.info("[{}] {}{}", traceId.getId(), addSpace(START_PREFIX,
traceId.getLevel()), message);

        return new TraceStatus(traceId, startTimeMs, message);
    }

    @Override
```

```
    public void end(TraceStatus status) {
        complete(status, null);
    }

    @Override
    public void exception(TraceStatus status, Exception e) {
        complete(status, e);
    }

    private void complete(TraceStatus status, Exception e) {
        Long stopTimeMs = System.currentTimeMillis();
        long resultTimeMs = stopTimeMs - status.getStartTimeMs();
        TraceId traceId = status.getTraceId();
        if (e == null) {
            log.info("[{}] {}{} time={}ms", traceId.getId(),
addSpace(COMPLETE_PREFIX, traceId.getLevel()), status.getMessage(),
resultTimeMs);
        } else {
            log.info("[{}] {}{} time={}ms ex={}", traceId.getId(),
addSpace(EX_PREFIX, traceId.getLevel()), status.getMessage(), resultTimeMs,
e.toString());
        }

        releaseTraceId();
    }

    private void syncTraceId() {
        if (traceIdHolder == null) {
            traceIdHolder = new TraceId();
        } else {
            traceIdHolder = traceIdHolder.createNextId();
        }
    }

    private void releaseTraceId() {
        if (traceIdHolder.isFirstLevel()) {
            traceIdHolder = null; //destroy
        } else {
            traceIdHolder = traceIdHolder.createPreviousId();
```

```
        }
    }

    private static String addSpace(String prefix, int level) {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < level; i++) {
            sb.append( (i == level - 1) ? "|" + prefix : "|   ");
        }
        return sb.toString();
    }
}
```

`FieldLogTrace`는 기존에 만들었던 `HelloTraceV2`와 거의 같은 기능을 한다.
`TraceId`를 동기화 하는 부분만 파라미터를 사용하는 것에서 `TraceId traceIdHolder` 필드를 사용하도록 변경되었다.
이제 직전 로그의 `TraceId`는 파라미터로 전달되는 것이 아니라 `FieldLogTrace`의 필드인 `traceIdHolder`에 저장된다.

여기서 중요한 부분은 로그를 시작할 때 호출하는 `syncTraceId()`와 로그를 종료할 때 호출하는 `releaseTraceId()`이다.

- `syncTraceId()`
  - `TraceId`를 새로 만들거나 앞선 로그의 `TraceId`를 참고해서 동기화하고, `level`도 증가한다.
  - 최초 호출이면 `TraceId`를 새로 만든다.
  - 직전 로그가 있으면 해당 로그의 `TraceId`를 참고해서 동기화하고, `level`도 하나 증가한다.
  - 결과를 `traceIdHolder`에 보관한다.
- `releaseTraceId()`
  - 메서드를 추가로 호출할 때는 `level`이 하나 증가해야 하지만, 메서드 호출이 끝나면 `level`이 하나 감소해야 한다.
  - `releaseTraceId()`는 `level`을 하나 감소한다.
  - 만약 최초 호출(`level==0`)이면 내부에서 관리하는 `traceId`를 제거한다.

```
[c80f5dbb] OrderController.request()       //syncTraceId(): 최초 호출 level=0
[c80f5dbb] |-->OrderService.orderItem()    //syncTraceId(): 직전 로그 있음 level=1
증가
[c80f5dbb] |   |-->OrderRepository.save()  //syncTraceId(): 직전 로그 있음 level=2
증가
[c80f5dbb] |   |<--OrderRepository.save() time=1005ms //releaseTraceId():
```

```
level=2->1 감소
[c80f5dbb] |<--OrderService.orderItem() time=1014ms   //releaseTraceId():
level=1->0 감소
[c80f5dbb] OrderController.request() time=1017ms      //releaseTraceId():
level==0, traceId 제거
```

테스트 코드를 통해서 실행해보자.

FieldLogTraceTest

```
package hello.advanced.trace.logtrace;

import hello.advanced.trace.TraceStatus;
import org.junit.jupiter.api.Test;

class FieldLogTraceTest {

    FieldLogTrace trace = new FieldLogTrace();

    @Test
    void begin_end_level2() {
        TraceStatus status1 = trace.begin("hello1");
        TraceStatus status2 = trace.begin("hello2");
        trace.end(status2);
        trace.end(status1);
    }

    @Test
    void begin_exception_level2() {
        TraceStatus status1 = trace.begin("hello");
        TraceStatus status2 = trace.begin("hello2");
        trace.exception(status2, new IllegalStateException());
        trace.exception(status1, new IllegalStateException());
    }

}
```

begin_end_level2() - 실행 결과

```
[ed72b67d] hello1
[ed72b67d] |-->hello2
[ed72b67d] |<--hello2 time=2ms
[ed72b67d] hello1 time=6ms
```

begin_exception_level2() - 실행 결과

```
[59770788] hello
[59770788] |-->hello2
[59770788] |<X-hello2 time=3ms ex=java.lang.IllegalStateException
[59770788] hello time=8ms ex=java.lang.IllegalStateException
```

실행 결과를 보면 `트랜잭션ID` 도 동일하게 나오고, `level` 을 통한 깊이도 잘 표현된다.
`FieldLogTrace.traceIdHolder` 필드를 사용해서 `TraceId` 가 잘 동기화 되는 것을 확인할 수 있다.
이제 불필요하게 `TraceId` 를 파라미터로 전달하지 않아도 되고, 애플리케이션의 메서드 파라미터도 변경하지 않아도 된다.

필드 동기화 - 적용

지금까지 만든 `FieldLogTrace` 를 애플리케이션에 적용해보자.

LogTrace 스프링 빈 등록

`FieldLogTrace` 를 수동으로 스프링 빈으로 등록하자. 수동으로 등록하면 향후 구현체를 편리하게 변경할 수 있다는 장점이 있다.

LogTraceConfig

```
package hello.advanced;

import hello.advanced.trace.logtrace.FieldLogTrace;
import hello.advanced.trace.logtrace.LogTrace;
```

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class LogTraceConfig {

    @Bean
    public LogTrace logTrace() {
        return new FieldLogTrace();
    }
}
```

v2 -> v3 복사

로그 추적기 V3를 적용하기 전에 먼저 기존 코드를 복사하자.

- `hello.advanced.app.v3` 패키지 생성
- 복사
  - `v2.OrderControllerV2` ➡ `v3.OrderControllerV3`
  - `v2.OrderServiceV2` ➡ `v3.OrderServiceV3`
  - `v2.OrderRepositoryV2` ➡ `v3.OrderRepositoryV3`
- 코드 내부 의존관계를 클래스를 V3으로 변경
  - `OrderControllerV3` : `OrderServiceV2` ➡ `OrderServiceV3`
  - `OrderServiceV3` : `OrderRepositoryV2` ➡ `OrderRepositoryV3`
- `OrderControllerV3` 매핑 정보 변경
  - `@GetMapping("/v3/request")`
- `HelloTraceV2` ➡ `LogTrace` 인터페이스 사용 ➡ **주의!**
- `TraceId traceId` 파라미터를 모두 제거
- `beginSync()` ➡ `begin`으로 사용하도록 변경

전체 코드는 다음과 같다.

OrderControllerV3

```
package hello.advanced.app.v3;
```

```
import hello.advanced.trace.TraceStatus;
import hello.advanced.trace.logtrace.LogTrace;
import lombok.RequiredArgsConstructor;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequiredArgsConstructor
public class OrderControllerV3 {

    private final OrderServiceV3 orderService;
    private final LogTrace trace;

    @GetMapping("/v3/request")
    public String request(String itemId) {

        TraceStatus status = null;
        try {
            status = trace.begin("OrderController.request()");
            orderService.orderItem(itemId);
            trace.end(status);
            return "ok";
        } catch (Exception e) {
            trace.exception(status, e);
            throw e; //예외를 꼭 다시 던져주어야 한다.
        }
    }

}
```

OrderServiceV3

```
package hello.advanced.app.v3;

import hello.advanced.trace.TraceId;
import hello.advanced.trace.TraceStatus;
import hello.advanced.trace.logtrace.LogTrace;
```

```
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;

@Service
@RequiredArgsConstructor
public class OrderServiceV3 {

    private final OrderRepositoryV3 orderRepository;
    private final LogTrace trace;

    public void orderItem(String itemId) {
        TraceStatus status = null;
        try {
            status = trace.begin("OrderService.orderItem()");
            orderRepository.save(itemId);
            trace.end(status);
        } catch (Exception e) {
            trace.exception(status, e);
            throw e;
        }
    }

}
```

OrderRepositoryV3

```
package hello.advanced.app.v3;

import hello.advanced.trace.TraceId;
import hello.advanced.trace.TraceStatus;
import hello.advanced.trace.logtrace.LogTrace;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Repository;

@Repository
@RequiredArgsConstructor
public class OrderRepositoryV3 {
```

```
    private final LogTrace trace;

    public void save(String itemId) {

        TraceStatus status = null;
        try {
            status = trace.begin("OrderRepository.save()");

            //저장 로직
            if (itemId.equals("ex")) {
                throw new IllegalStateException("예외 발생!");
            }
            sleep(1000);

            trace.end(status);
        } catch (Exception e) {
            trace.exception(status, e);
            throw e;
        }

    }

    private void sleep(int millis) {
        try {
            Thread.sleep(millis);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

정상 실행

- http://localhost:8080/v3/request?itemId=hello

정상 실행 로그

```
[f8477cfc] OrderController.request()
[f8477cfc] |-->OrderService.orderItem()
[f8477cfc] |   |-->OrderRepository.save()
[f8477cfc] |   |<--OrderRepository.save() time=1004ms
[f8477cfc] |<--OrderService.orderItem() time=1006ms
[f8477cfc] OrderController.request() time=1007ms
```

예외 실행

- http://localhost:8080/v3/request?itemId=ex

예외 실행 로그

```
[c426fcfc] OrderController.request()
[c426fcfc] |-->OrderService.orderItem()
[c426fcfc] |   |-->OrderRepository.save()
[c426fcfc] |   |<X-OrderRepository.save() time=0ms
ex=java.lang.IllegalStateException: 예외 발생!
[c426fcfc] |<X-OrderService.orderItem() time=7ms
ex=java.lang.IllegalStateException: 예외 발생!
[c426fcfc] OrderController.request() time=7ms
ex=java.lang.IllegalStateException: 예외 발생!
```

`traceIdHolder` 필드를 사용한 덕분에 파라미터 추가 없는 깔끔한 로그 추적기를 완성했다. 이제 실제 서비스에 배포한다고 가정해보자.

필드 동기화 - 동시성 문제

잘 만든 로그 추적기를 실제 서비스에 배포했다 가정해보자.
테스트 할 때는 문제가 없는 것 처럼 보인다. 사실 직전에 만든 `FieldLogTrace`는 심각한 동시성 문제를 가지고 있다.
동시성 문제를 확인하려면 다음과 같이 동시에 여러번 호출해보면 된다.

동시성 문제 확인
다음 로직을 1초 안에 2번 실행해보자.

- http://localhost:8080/v3/request?itemId=hello
- http://localhost:8080/v3/request?itemId=hello

기대하는 결과

```
[nio-8080-exec-3] [52808e46] OrderController.request()
[nio-8080-exec-3] [52808e46] |-->OrderService.orderItem()
[nio-8080-exec-3] [52808e46] |   |-->OrderRepository.save()
[nio-8080-exec-4] [4568423c] OrderController.request()
[nio-8080-exec-4] [4568423c] |-->OrderService.orderItem()
[nio-8080-exec-4] [4568423c] |   |-->OrderRepository.save()
[nio-8080-exec-3] [52808e46] |   |<--OrderRepository.save() time=1001ms
[nio-8080-exec-3] [52808e46] |<--OrderService.orderItem() time=1001ms
[nio-8080-exec-3] [52808e46] OrderController.request() time=1003ms
[nio-8080-exec-4] [4568423c] |   |<--OrderRepository.save() time=1000ms
[nio-8080-exec-4] [4568423c] |<--OrderService.orderItem() time=1001ms
[nio-8080-exec-4] [4568423c] OrderController.request() time=1001ms
```

동시에 여러 사용자가 요청하면 여러 쓰레드가 동시에 애플리케이션 로직을 호출하게 된다. 따라서 로그는 이렇게 섞여서 출력된다.

기대하는 결과 - 로그 분리해서 확인하기

```
[52808e46]
[nio-8080-exec-3] [52808e46] OrderController.request()
[nio-8080-exec-3] [52808e46] |-->OrderService.orderItem()
[nio-8080-exec-3] [52808e46] |   |-->OrderRepository.save()
[nio-8080-exec-3] [52808e46] |   |<--OrderRepository.save() time=1001ms
[nio-8080-exec-3] [52808e46] |<--OrderService.orderItem() time=1001ms
[nio-8080-exec-3] [52808e46] OrderController.request() time=1003ms

[4568423c]
[nio-8080-exec-4] [4568423c] OrderController.request()
[nio-8080-exec-4] [4568423c] |-->OrderService.orderItem()
[nio-8080-exec-4] [4568423c] |   |-->OrderRepository.save()
[nio-8080-exec-4] [4568423c] |   |<--OrderRepository.save() time=1000ms
[nio-8080-exec-4] [4568423c] |<--OrderService.orderItem() time=1001ms
[nio-8080-exec-4] [4568423c] OrderController.request() time=1001ms
```

로그가 섞여서 출력되더라도 특정 트랜잭션ID로 구분해서 직접 분류해보면 이렇게 깔끔하게 분리된 것을 확인할 수 있다. 그런데 실제 결과는 기대한 것과 다르게 다음과 같이 출력된다.

실제 결과

```
[nio-8080-exec-3] [aaaaaaaa] OrderController.request()
[nio-8080-exec-3] [aaaaaaaa] |-->OrderService.orderItem()
[nio-8080-exec-3] [aaaaaaaa] |   |-->OrderRepository.save()
[nio-8080-exec-4] [aaaaaaaa] |   |   |-->OrderController.request()
[nio-8080-exec-4] [aaaaaaaa] |   |   |   |-->OrderService.orderItem()
[nio-8080-exec-4] [aaaaaaaa] |   |   |   |   |-->OrderRepository.save()
[nio-8080-exec-3] [aaaaaaaa] |   |<--OrderRepository.save() time=1005ms
[nio-8080-exec-3] [aaaaaaaa] |<--OrderService.orderItem() time=1005ms
[nio-8080-exec-3] [aaaaaaaa] OrderController.request() time=1005ms
[nio-8080-exec-4] [aaaaaaaa] |   |   |   |   |<--OrderRepository.save()
time=1005ms
[nio-8080-exec-4] [aaaaaaaa] |   |   |   |<--OrderService.orderItem()
time=1005ms
[nio-8080-exec-4] [aaaaaaaa] |   |   |<--OrderController.request() time=1005ms
```

실제 결과 - 로그 분리해서 확인하기

```
[nio-8080-exec-3]
[nio-8080-exec-3] [aaaaaaaa] OrderController.request()
[nio-8080-exec-3] [aaaaaaaa] |-->OrderService.orderItem()
[nio-8080-exec-3] [aaaaaaaa] |   |-->OrderRepository.save()
[nio-8080-exec-3] [aaaaaaaa] |   |<--OrderRepository.save() time=1005ms
[nio-8080-exec-3] [aaaaaaaa] |<--OrderService.orderItem() time=1005ms
[nio-8080-exec-3] [aaaaaaaa] OrderController.request() time=1005ms

[nio-8080-exec-4]
[nio-8080-exec-4] [aaaaaaaa] |   |   |-->OrderController.request()
[nio-8080-exec-4] [aaaaaaaa] |   |   |   |-->OrderService.orderItem()
[nio-8080-exec-4] [aaaaaaaa] |   |   |   |   |-->OrderRepository.save()
[nio-8080-exec-4] [aaaaaaaa] |   |   |   |   |<--OrderRepository.save()
time=1005ms
```

```
[nio-8080-exec-4] [aaaaaaaa] |   |   |   |<--OrderService.orderItem()
time=1005ms
[nio-8080-exec-4] [aaaaaaaa] |   |   |<--OrderController.request() time=1005ms
```

기대한 것과 전혀 다른 문제가 발생한다. `트랜잭션ID` 도 동일하고, `level` 도 뭔가 많이 꼬인 것 같다. 분명히 테스트 코드로 작성할 때는 문제가 없었는데, 무엇이 문제일까?

동시성 문제
사실 이 문제는 동시성 문제이다.
`FieldLogTrace` 는 싱글톤으로 등록된 스프링 빈이다. 이 객체의 인스턴스가 애플리케이션에 딱 1 존재한다는 뜻이다. 이렇게 하나만 있는 인스턴스의 `FieldLogTrace.traceIdHolder` 필드를 여러 쓰레드가 동시에 접근하기 때문에 문제가 발생한다.
실무에서 한번 나타나면 개발자를 가장 괴롭히는 문제도 바로 이러한 동시성 문제이다.

동시성 문제 - 예제 코드

동시성 문제가 어떻게 발생하는지 단순화해서 알아보자.

테스트에서도 lombok을 사용하기 위해 다음 코드를 추가하자.
`build.gradle`

```
dependencies {
    ...
    //테스트에서 lombok 사용
    testCompileOnly 'org.projectlombok:lombok'
    testAnnotationProcessor 'org.projectlombok:lombok'
}
```

이렇게 해야 테스트 코드에서 `@Slfj4` 같은 애노테이션이 작동한다.

FieldService
주의: 테스트 코드(src/test)에 위치한다.

```
package hello.advanced.trace.threadlocal.code;

import lombok.extern.slf4j.Slf4j;
```

```
@Slf4j
public class FieldService {

    private String nameStore;

    public String logic(String name) {
        log.info("저장 name={} -> nameStore={}", name, nameStore);
        nameStore = name;
        sleep(1000);
        log.info("조회 nameStore={}",nameStore);
        return nameStore;
    }

    private void sleep(int millis) {
        try {
            Thread.sleep(millis);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

}
```

매우 단순한 로직이다. 파라미터로 넘어온 `name`을 필드인 `nameStore`에 저장한다. 그리고 1초간 쉰 다음 필드에 저장된 `nameStore`를 반환한다.

FieldServiceTest

```
package hello.advanced.trace.threadlocal;

import hello.advanced.trace.threadlocal.code.FieldService;
import lombok.extern.slf4j.Slf4j;
import org.junit.jupiter.api.Test;

@Slf4j
public class FieldServiceTest {

    private FieldService fieldService = new FieldService();
```

```
    @Test
    void field() {
        log.info("main start");
        Runnable userA = () -> {
            fieldService.logic("userA");
        };

        Runnable userB = () -> {
            fieldService.logic("userB");
        };

        Thread threadA = new Thread(userA);
        threadA.setName("thread-A");
        Thread threadB = new Thread(userB);
        threadB.setName("thread-B");

        threadA.start(); //A실행
        sleep(2000); //동시성 문제 발생X
//        sleep(100); //동시성 문제 발생O
        threadB.start(); //B실행

        sleep(3000); //메인 쓰레드 종료 대기
        log.info("main exit");
    }

    private void sleep(int millis) {
        try {
            Thread.sleep(millis);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

순서대로 실행

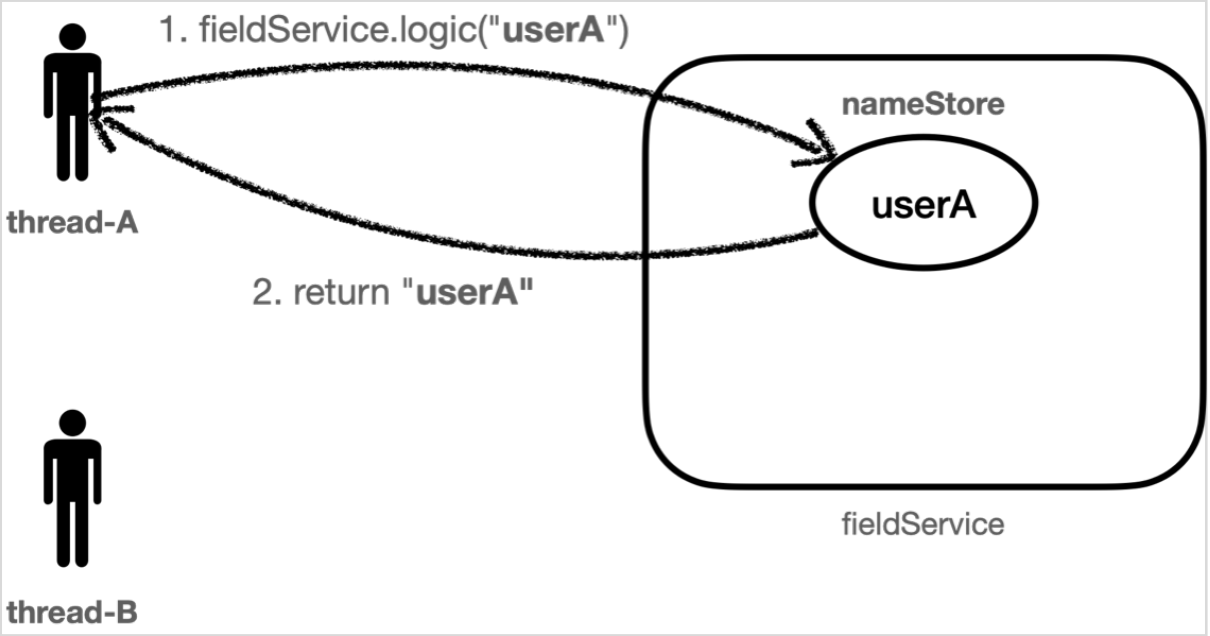
`sleep(2000)`을 설정해서 `thread-A`의 실행이 끝나고 나서 `thread-B`가 실행되도록 해보자.
참고로 `FieldService.logic()` 메서드는 내부에 `sleep(1000)`으로 1초의 지연이 있다. 따라서 1초

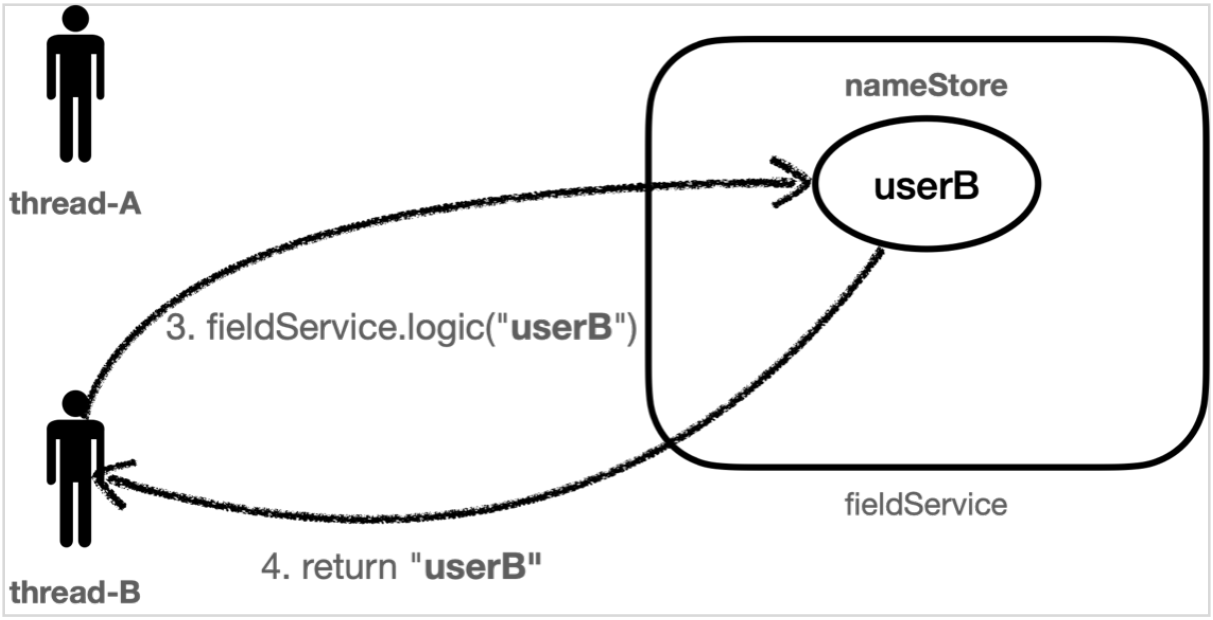
이후에 호출하면 순서대로 실행할 수 있다. 여기서는 넉넉하게 2초 (2000ms)를 설정했다.

```
sleep(2000); //동시성 문제 발생X
//sleep(100); //동시성 문제 발생O
```

실행 결과

```
[Test worker] main start
[Thread-A]    저장 name=userA -> nameStore=null
[Thread-A]    조회 nameStore=userA
[Thread-B]    저장 name=userB -> nameStore=userA
[Thread-B]    조회 nameStore=userB
[Test worker] main exit
```





실행 결과를 보면 문제가 없다.

- `Thread-A` 는 `userA` 를 `nameStore` 에 저장했다.
- `Thread-A` 는 `userA` 를 `nameStore` 에서 조회했다.
- `Thread-B` 는 `userB` 를 `nameStore` 에 저장했다.
- `Thread-B` 는 `userB` 를 `nameStore` 에서 조회했다.

동시성 문제 발생 코드

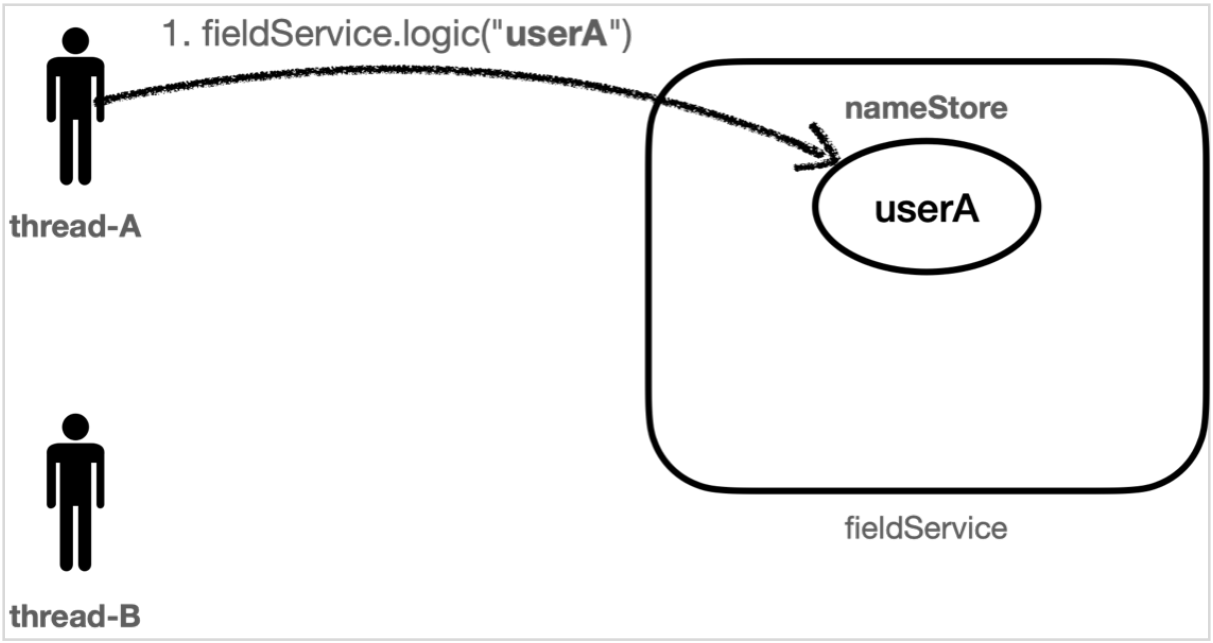
이번에는 `sleep(100)` 을 설정해서 `thread-A` 의 작업이 끝나기 전에 `thread-B` 가 실행되도록 해보자. 참고로 `FieldService.logic()` 메서드는 내부에 `sleep(1000)` 으로 1초의 지연이 있다. 따라서 1초 이후에 호출하면 순서대로 실행할 수 있다. 다음에 설정할 100(ms)는 0.1초이기 때문에 `thread-A` 의 작업이 끝나기 전에 `thread-B` 가 실행된다.

```
//sleep(2000); //동시성 문제 발생X
sleep(100); //동시성 문제 발생O
```

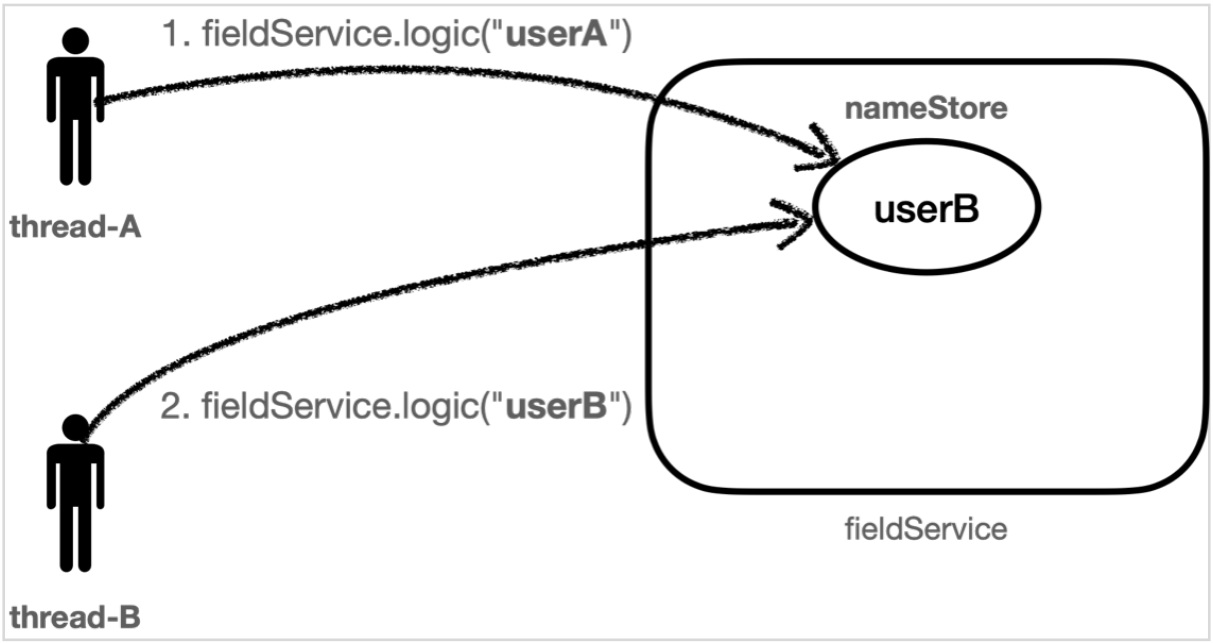
실행 결과

```
[Test worker] main start
[Thread-A]    저장 name=userA -> nameStore=null
[Thread-B]    저장 name=userB -> nameStore=userA
[Thread-A]    조회 nameStore=userB
[Thread-B]    조회 nameStore=userB
[Test worker] main exit
```

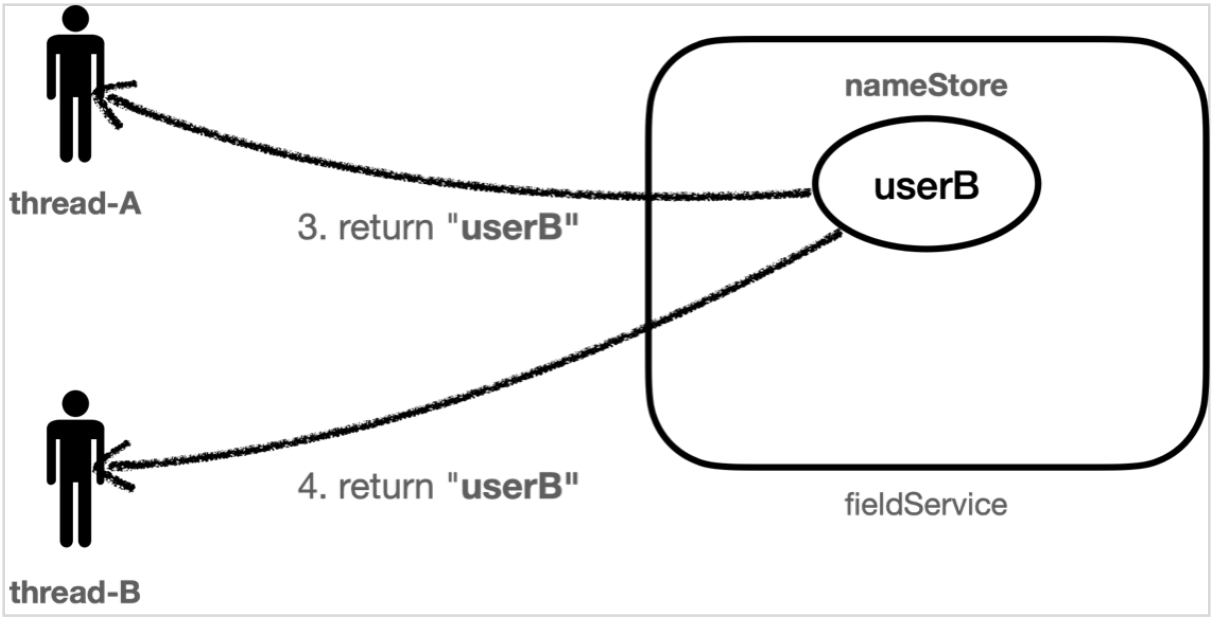
실행 결과를 보자. 저장하는 부분은 문제가 없다. 문제는 조회하는 부분에서 발생한다.



- 먼저 `thread-A` 가 `userA` 값을 `nameStore` 에 보관한다.



- 0.1초 이후에 `thread-B` 가 `userB` 의 값을 `nameStore` 에 보관한다. 기존에 `nameStore` 에 보관되어 있던 `userA` 값은 제거되고 `userB` 값이 저장된다.



- `thread-A`의 호출이 끝나면서 `nameStore`의 결과를 반환받는데, 이때 `nameStore`는 앞의 2번에서 `userB`의 값으로 대체되었다. 따라서 기대했던 `userA`의 값이 아니라 `userB`의 값이 반환된다.
- `thread-B`의 호출이 끝나면서 `nameStore`의 결과인 `userB`를 반환받는다.

정리하면 다음과 같다.

- 1. `Thread-A`는 `userA`를 `nameStore`에 저장했다.
- 2. `Thread-B`는 `userB`를 `nameStore`에 저장했다.
- 3. `Thread-A`는 `userB`를 `nameStore`에서 조회했다.
- 4. `Thread-B`는 `userB`를 `nameStore`에서 조회했다.

동시성 문제

결과적으로 `Thread-A` 입장에서는 저장한 데이터와 조회한 데이터가 다른 문제가 발생한다. 이처럼 여러 쓰레드가 동시에 같은 인스턴스의 필드 값을 변경하면서 발생하는 문제를 동시성 문제라 한다. 이런 동시성 문제는 여러 쓰레드가 같은 인스턴스의 필드에 접근해야 하기 때문에 트래픽이 적은 상황에서는 확률상 잘 나타나지 않고, 트래픽이 점점 많아질 수 록 자주 발생한다.

특히 스프링 빈 처럼 싱글톤 객체의 필드를 변경하며 사용할 때 이러한 동시성 문제를 조심해야 한다.

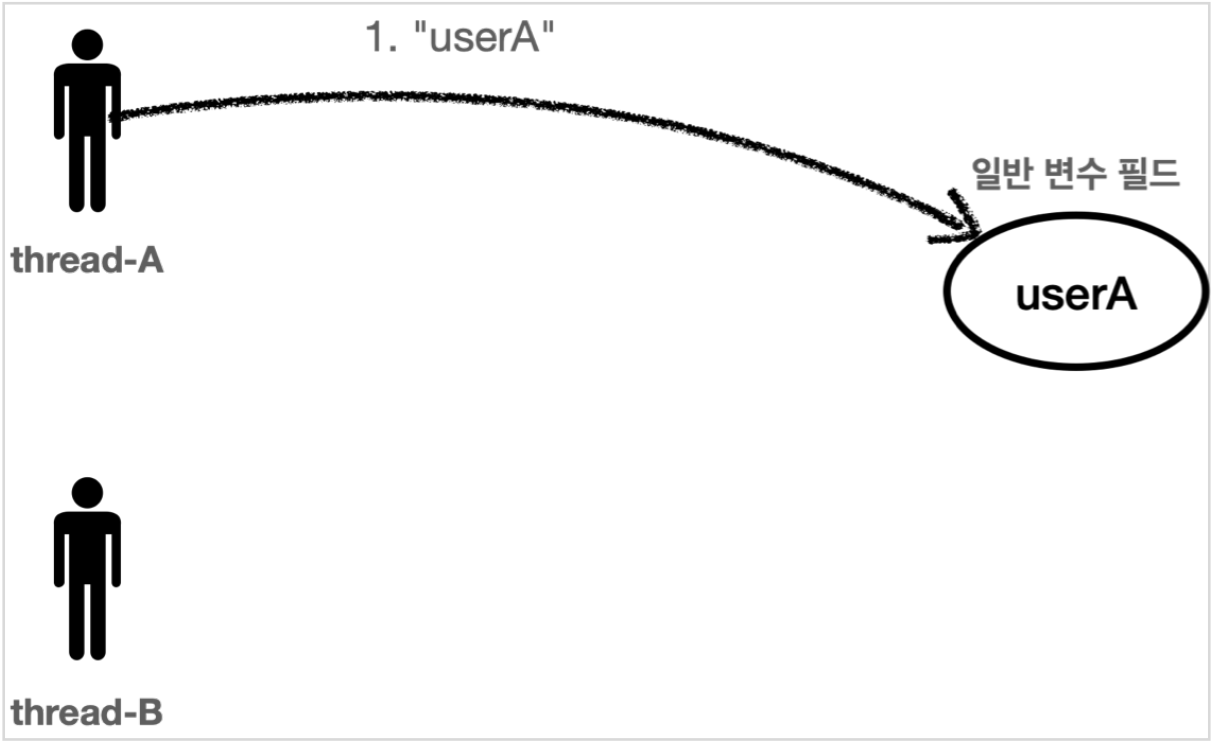
> **참고**
>
> 이런 동시성 문제는 지역 변수에서는 발생하지 않는다. 지역 변수는 쓰레드마다 각각 다른 메모리 영역이 할당된다.
>
> 동시성 문제가 발생하는 곳은 같은 인스턴스의 필드(주로 싱글톤에서 자주 발생), 또는 static 같은 공용 필드에 접근할 때 발생한다.
>
> 동시성 문제는 값을 읽기만 하면 발생하지 않는다. 어디선가 값을 변경하기 때문에 발생한다.

그렇다면 지금처럼 싱글톤 객체의 필드를 사용하면서 동시성 문제를 해결하려면 어떻게 해야할까? 다시 파라미터를 전달하는 방식으로 돌아가야 할까? 이럴 때 사용하는 것이 바로 쓰레드 로컬이다.

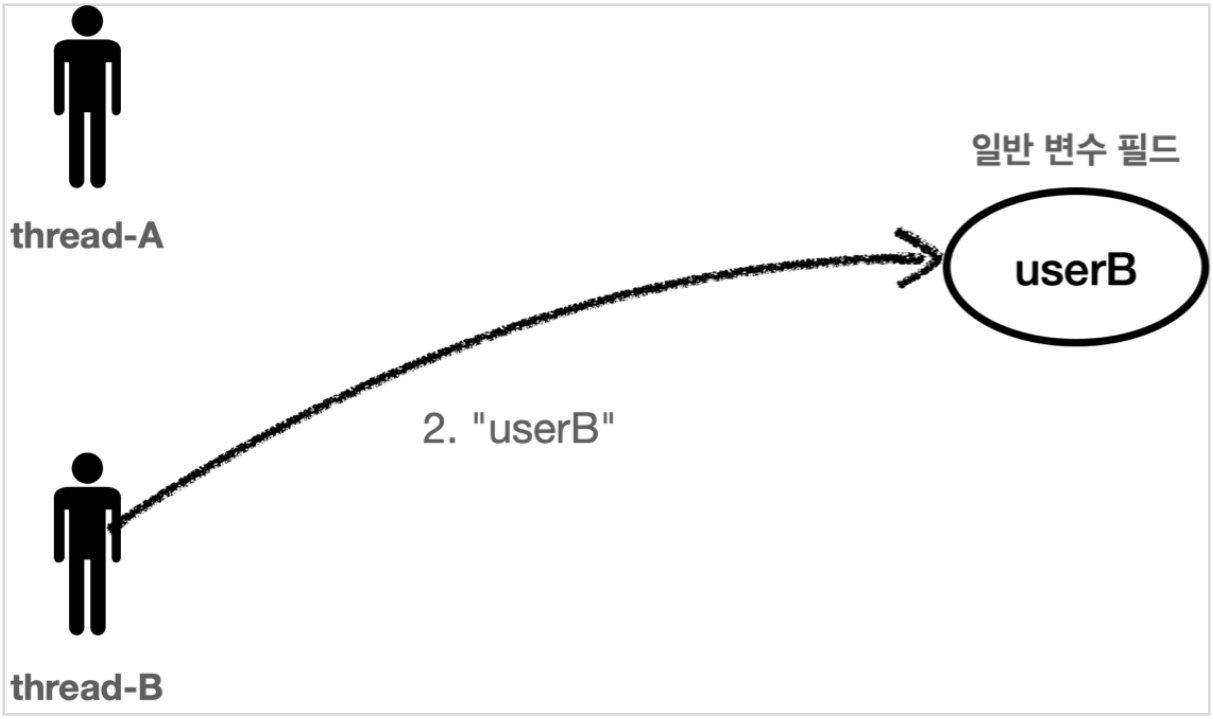
ThreadLocal - 소개

쓰레드 로컬은 해당 쓰레드만 접근할 수 있는 특별한 저장소를 말한다. 쉽게 이야기해서 물건 보관 창구를 떠올리면 된다. 여러 사람이 같은 물건 보관 창구를 사용하더라도 창구 직원은 사용자를 인식해서 사용자별로 확실하게 물건을 구분해준다.
사용자A, 사용자B 모두 창구 직원을 통해서 물건을 보관하고, 꺼내지만 창구 지원이 사용자에 따라 보관한 물건을 구분해주는 것이다.

일반적인 변수 필드
여러 쓰레드가 같은 인스턴스의 필드에 접근하면 처음 쓰레드가 보관한 데이터가 사라질 수 있다.



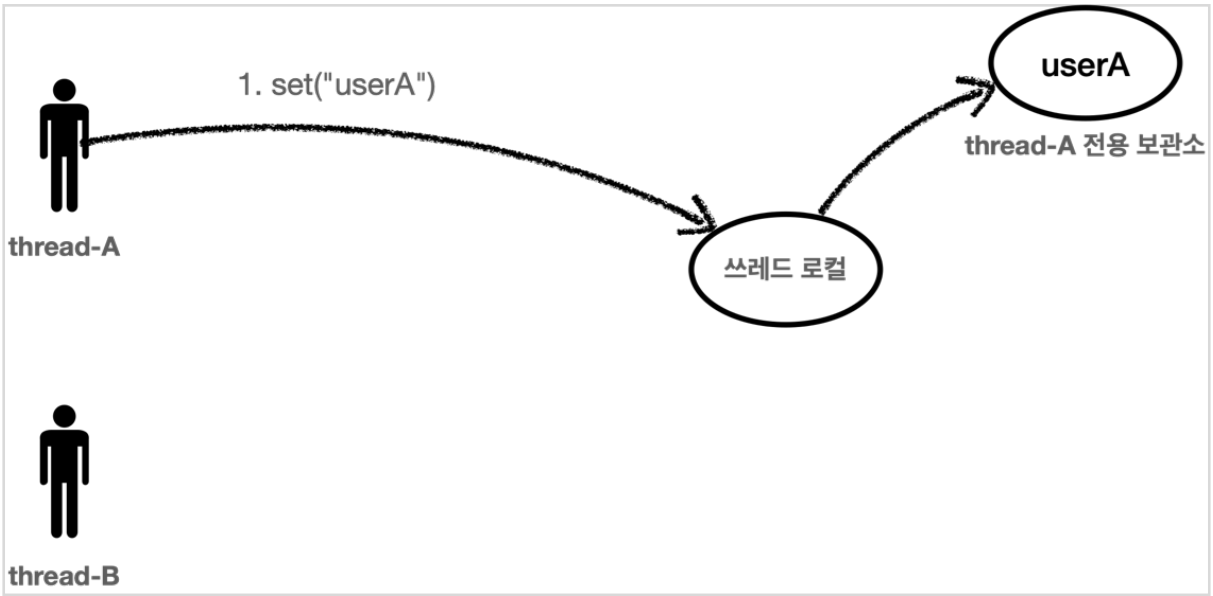
`thread-A`가 `userA`라는 값을 저장하고



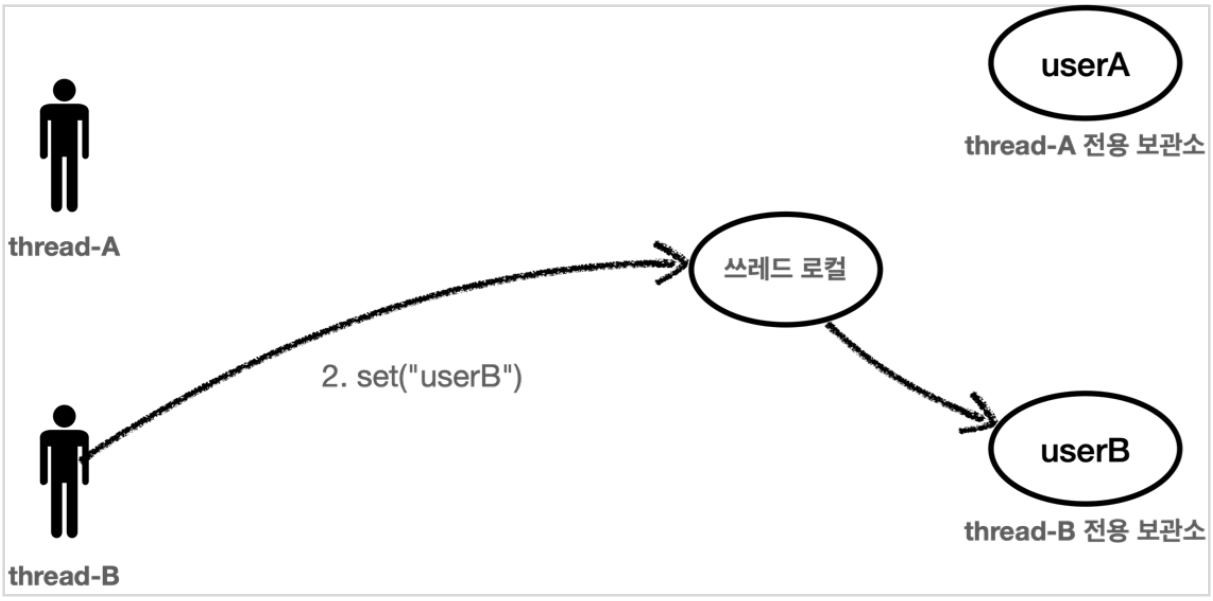
`thread-B`가 `userB`라는 값을 저장하면 직전에 `thread-A`가 저장한 `userA` 값은 사라진다.

쓰레드 로컬

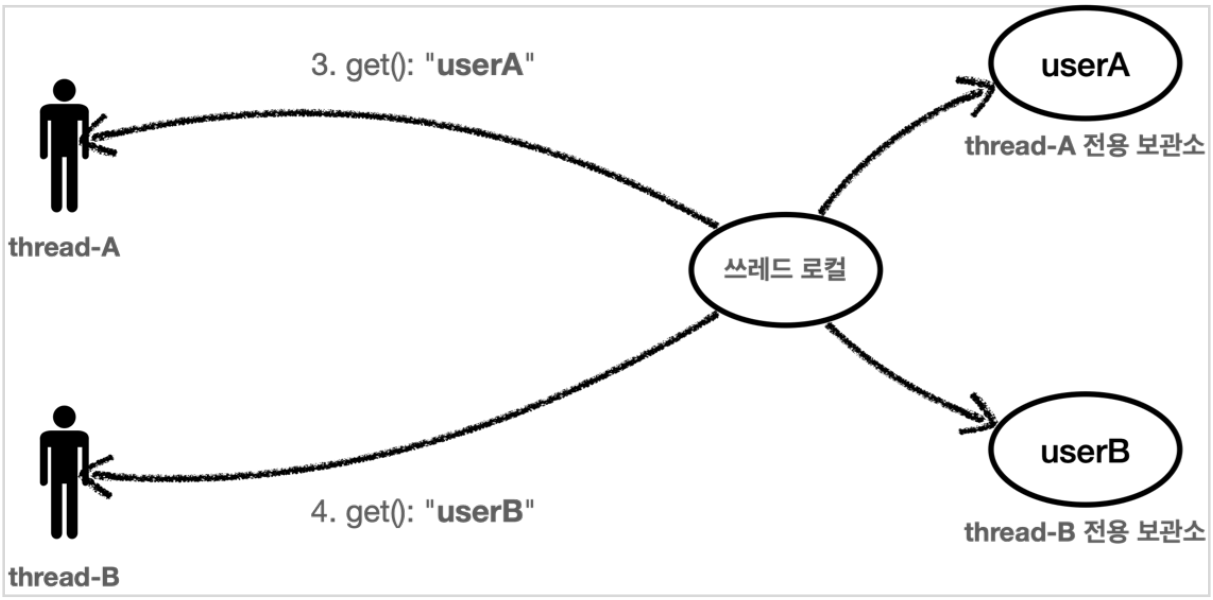
쓰레드 로컬을 사용하면 각 쓰레드마다 별도의 내부 저장소를 제공한다. 따라서 같은 인스턴스의 쓰레드 로컬 필드에 접근해도 문제 없다.



`thread-A`가 `userA`라는 값을 저장하면 쓰레드 로컬은 `thread-A` 전용 보관소에 데이터를 안전하게 보관한다.



`thread-B`가 `userB`라는 값을 저장하면 쓰레드 로컬은 `thread-B` 전용 보관소에 데이터를 안전하게 보관한다.



쓰레드 로컬을 통해서 데이터를 조회할 때도 `thread-A`가 조회하면 쓰레드 로컬은 `thread-A` 전용 보관소에서 `userA` 데이터를 반환해준다. 물론 `thread-B`가 조회하면 `thread-B` 전용 보관소에서 `userB` 데이터를 반환해준다.

자바는 언어차원에서 쓰레드 로컬을 지원하기 위한 `java.lang.ThreadLocal` 클래스를 제공한다.

ThreadLocal - 예제 코드

예제 코드를 통해서 `ThreadLocal`을 학습해보자.

ThreadLocalService

주의: 테스트 코드(src/test)에 위치한다.

```
package hello.advanced.trace.threadlocal.code;

import lombok.extern.slf4j.Slf4j;

@Slf4j
public class ThreadLocalService {

    private ThreadLocal<String> nameStore = new ThreadLocal<>();

    public String logic(String name) {
        log.info("저장 name={} -> nameStore={}", name, nameStore.get());
        nameStore.set(name);
        sleep(1000);
        log.info("조회 nameStore={}",nameStore.get());
        return nameStore.get();
    }

    private void sleep(int millis) {
        try {
            Thread.sleep(millis);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

}
```

기존에 있던 `FieldService`와 거의 같은 코드인데, `nameStore` 필드가 일반 `String` 타입에서 `ThreadLocal`을 사용하도록 변경되었다.

ThreadLocal 사용법

- 값 저장: `ThreadLocal.set(xxx)`
- 값 조회: `ThreadLocal.get()`
- 값 제거: `ThreadLocal.remove()`

> **주의**
> 해당 쓰레드가 쓰레드 로컬을 모두 사용하고 나면 `ThreadLocal.remove()` 를 호출해서 쓰레드 로컬에 저장된 값을 제거해주어야 한다. 제거하는 구체적인 예제는 조금 뒤에 설명하겠다.

ThreadLocalServiceTest

```
package hello.advanced.trace.threadlocal;

import hello.advanced.trace.threadlocal.code.ThreadLocalService;
import lombok.extern.slf4j.Slf4j;
import org.junit.jupiter.api.Test;

@Slf4j
public class ThreadLocalServiceTest {

    private ThreadLocalService service = new ThreadLocalService();

    @Test
    void threadLocal() {
        log.info("main start");
        Runnable userA = () -> {
            service.logic("userA");
        };

        Runnable userB = () -> {
            service.logic("userB");
        };

        Thread threadA = new Thread(userA);
        threadA.setName("thread-A");
        Thread threadB = new Thread(userB);
        threadB.setName("thread-B");

        threadA.start();
        sleep(100);
        threadB.start();

        sleep(2000);
        log.info("main exit");
```

```
    }

    private void sleep(int millis) {
        try {
            Thread.sleep(millis);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

실행 결과

```
[Test worker] main start
[Thread-A]    저장 name=userA -> nameStore=null
[Thread-B]    저장 name=userB -> nameStore=null
[Thread-A]    조회 nameStore=userA
[Thread-B]    조회 nameStore=userB
[Test worker] main exit
```

쓰레드 로컬 덕분에 쓰레드 마다 각각 별도의 데이터 저장소를 가지게 되었다. 결과적으로 동시성 문제도 해결되었다.

쓰레드 로컬 동기화 - 개발

`FieldLogTrace`에서 발생했던 동시성 문제를 `ThreadLocal`로 해결해보자.
`TraceId traceIdHolder` 필드를 쓰레드 로컬을 사용하도록 `ThreadLocal<TraceId> traceIdHolder`로 변경하면 된다.

필드 대신에 쓰레드 로컬을 사용해서 데이터를 동기화하는 `ThreadLocalLogTrace`를 새로 만들자.

ThreadLocalLogTrace

```
package hello.advanced.trace.logtrace;
```

```
import hello.advanced.trace.TraceId;
import hello.advanced.trace.TraceStatus;
import lombok.extern.slf4j.Slf4j;

@Slf4j
public class ThreadLocalLogTrace implements LogTrace {

    private static final String START_PREFIX = "-->";
    private static final String COMPLETE_PREFIX = "<--";
    private static final String EX_PREFIX = "<X-";

    private ThreadLocal<TraceId> traceIdHolder = new ThreadLocal<>();

    @Override
    public TraceStatus begin(String message) {
        syncTraceId();
        TraceId traceId = traceIdHolder.get();
        Long startTimeMs = System.currentTimeMillis();
        log.info("[{}] {}{}", traceId.getId(), addSpace(START_PREFIX,
traceId.getLevel()), message);

        return new TraceStatus(traceId, startTimeMs, message);
    }

    @Override
    public void end(TraceStatus status) {
        complete(status, null);
    }

    @Override
    public void exception(TraceStatus status, Exception e) {
        complete(status, e);
    }

    private void complete(TraceStatus status, Exception e) {
        Long stopTimeMs = System.currentTimeMillis();
        long resultTimeMs = stopTimeMs - status.getStartTimeMs();
        TraceId traceId = status.getTraceId();
```

```
        if (e == null) {
            log.info("[{}] {}{} time={}ms", traceId.getId(),
addSpace(COMPLETE_PREFIX, traceId.getLevel()), status.getMessage(),
resultTimeMs);
        } else {
            log.info("[{}] {}{} time={}ms ex={}", traceId.getId(),
addSpace(EX_PREFIX, traceId.getLevel()), status.getMessage(), resultTimeMs,
e.toString());
        }

        releaseTraceId();
    }

    private void syncTraceId() {
        TraceId traceId = traceIdHolder.get();
        if (traceId == null) {
            traceIdHolder.set(new TraceId());
        } else {
            traceIdHolder.set(traceId.createNextId());
        }
    }

    private void releaseTraceId() {
        TraceId traceId = traceIdHolder.get();
        if (traceId.isFirstLevel()) {
            traceIdHolder.remove();//destroy
        } else {
            traceIdHolder.set(traceId.createPreviousId());
        }
    }

    private static String addSpace(String prefix, int level) {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < level; i++) {
            sb.append( (i == level - 1) ? "|" + prefix : "|   ");
        }
        return sb.toString();
    }
}
```

`traceIdHolder`가 필드에서 `ThreadLocal`로 변경되었다. 따라서 값을 저장할 때는 `set(..)`을 사용하고, 값을 조회할 때는 `get()`을 사용한다.

ThreadLocal.remove()

추가로 쓰레드 로컬을 모두 사용하고 나면 꼭 `ThreadLocal.remove()`를 호출해서 쓰레드 로컬에 저장된 값을 제거해주어야 한다.
쉽게 이야기해서 다음의 마지막 로그를 출력하고 나면 쓰레드 로컬의 값을 제거해야 한다.

```
[3f902f0b] hello1
[3f902f0b] |-->hello2
[3f902f0b] |<--hello2 time=2ms
[3f902f0b] hello1 time=6ms //end() -> releaseTraceId() -> level==0,
ThreadLocal.remove() 호출
```

여기서는 `releaseTraceId()`를 통해 `level`이 점점 낮아져서 2 ➡ 1 ➡ 0이 되면 로그를 처음 호출한 부분으로 돌아온 것이다. 따라서 이 경우 연관된 로그 출력이 끝난 것이다. 이제 더 이상 `TraceId` 값을 추적하지 않아도 된다. 그래서 `traceId.isFirstLevel()`(`level==0`)인 경우 `ThreadLocal.remove()`를 호출해서 쓰레드 로컬에 저장된 값을 제거해준다.

코드에 문제가 없는지 간단한 테스트를 만들어서 확인해보자.

ThreadLocalLogTraceTest

```
package hello.advanced.trace.logtrace;

import hello.advanced.trace.TraceStatus;
import lombok.extern.slf4j.Slf4j;
import org.junit.jupiter.api.Test;

@Slf4j
class ThreadLocalLogTraceTest {

    ThreadLocalLogTrace trace = new ThreadLocalLogTrace();

    @Test
```

```
    void begin_end_level2() {
        TraceStatus status1 = trace.begin("hello1");
        TraceStatus status2 = trace.begin("hello2");
        trace.end(status2);
        trace.end(status1);
    }

    @Test
    void begin_exception_level2() {
        TraceStatus status1 = trace.begin("hello");
        TraceStatus status2 = trace.begin("hello2");
        trace.exception(status2, new IllegalStateException());
        trace.exception(status1, new IllegalStateException());
    }

}
```

begin_end_level2() - 실행 결과

```
[3f902f0b] hello1
[3f902f0b] |-->hello2
[3f902f0b] |<--hello2 time=2ms
[3f902f0b] hello1 time=6ms
```

begin_exception_level2() - 실행 결과

```
[3dd9e4f1] hello
[3dd9e4f1] |-->hello2
[3dd9e4f1] |<X-hello2 time=3ms ex=java.lang.IllegalStateException
[3dd9e4f1] hello time=8ms ex=java.lang.IllegalStateException
```

멀티쓰레드 상황에서 문제가 없는지는 애플리케이션에 `ThreadLocalLogTrace`를 적용해서 확인해보자.

쓰레드 로컬 동기화 - 적용

LogTraceConfig - 수정

```
package hello.advanced;

import hello.advanced.trace.logtrace.LogTrace;
import hello.advanced.trace.logtrace.ThreadLocalLogTrace;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class LogTraceConfig {

    @Bean
    public LogTrace logTrace() {
        //return new FieldLogTrace();
        return new ThreadLocalLogTrace();
    }
}
```

동시성 문제가 있는 `FieldLogTrace` 대신에 문제를 해결한 `ThreadLocalLogTrace` 를 스프링 빈으로 등록하자.

정상 실행

- http://localhost:8080/v3/request?itemId=hello

정상 실행 로그

```
[f8477cfc] OrderController.request()
[f8477cfc] |-->OrderService.orderItem()
[f8477cfc] |   |-->OrderRepository.save()
[f8477cfc] |   |<--OrderRepository.save() time=1004ms
[f8477cfc] |<--OrderService.orderItem() time=1006ms
[f8477cfc] OrderController.request() time=1007ms
```

예외 실행

- http://localhost:8080/v3/request?itemId=ex

예외 실행 로그

```
[c426fcfc] OrderController.request()
[c426fcfc] |-->OrderService.orderItem()
[c426fcfc] |   |-->OrderRepository.save()
[c426fcfc] |   |<X-OrderRepository.save() time=0ms
ex=java.lang.IllegalStateException: 예외 발생!
[c426fcfc] |<X-OrderService.orderItem() time=7ms
ex=java.lang.IllegalStateException: 예외 발생!
[c426fcfc] OrderController.request() time=7ms
ex=java.lang.IllegalStateException: 예외 발생!
```

동시 요청

동시성 문제 확인

다음 로직을 1초 안에 2번 실행해보자.

- http://localhost:8080/v3/request?itemId=hello
- http://localhost:8080/v3/request?itemId=hello

실행 결과

```
[nio-8080-exec-3] [52808e46] OrderController.request()
[nio-8080-exec-3] [52808e46] |-->OrderService.orderItem()
[nio-8080-exec-3] [52808e46] |   |-->OrderRepository.save()
[nio-8080-exec-4] [4568423c] OrderController.request()
[nio-8080-exec-4] [4568423c] |-->OrderService.orderItem()
[nio-8080-exec-4] [4568423c] |   |-->OrderRepository.save()
[nio-8080-exec-3] [52808e46] |   |<--OrderRepository.save() time=1001ms
[nio-8080-exec-3] [52808e46] |<--OrderService.orderItem() time=1001ms
[nio-8080-exec-3] [52808e46] OrderController.request() time=1003ms
[nio-8080-exec-4] [4568423c] |   |<--OrderRepository.save() time=1000ms
[nio-8080-exec-4] [4568423c] |<--OrderService.orderItem() time=1001ms
[nio-8080-exec-4] [4568423c] OrderController.request() time=1001ms
```

로그 분리해서 확인하기

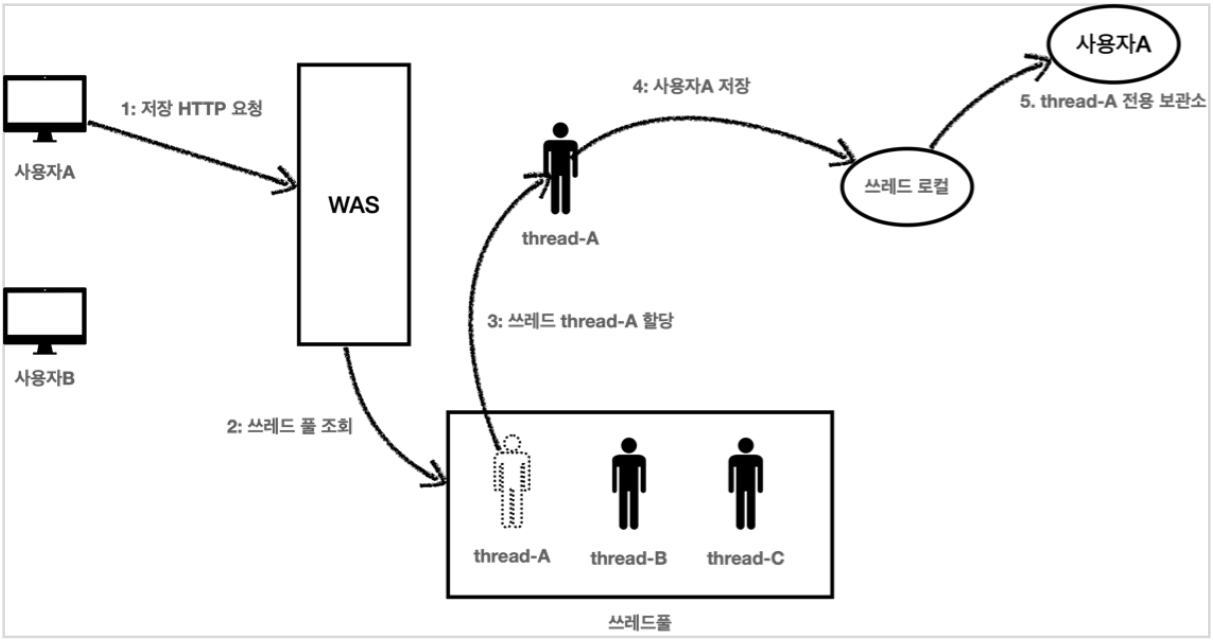
```
[nio-8080-exec-3]
[nio-8080-exec-3] [52808e46] OrderController.request()
[nio-8080-exec-3] [52808e46] |-->OrderService.orderItem()
[nio-8080-exec-3] [52808e46] |   |-->OrderRepository.save()
[nio-8080-exec-3] [52808e46] |   |<--OrderRepository.save() time=1001ms
[nio-8080-exec-3] [52808e46] |<--OrderService.orderItem() time=1001ms
[nio-8080-exec-3] [52808e46] OrderController.request() time=1003ms

[nio-8080-exec-4]
[nio-8080-exec-4] [4568423c] OrderController.request()
[nio-8080-exec-4] [4568423c] |-->OrderService.orderItem()
[nio-8080-exec-4] [4568423c] |   |-->OrderRepository.save()
[nio-8080-exec-4] [4568423c] |   |<--OrderRepository.save() time=1000ms
[nio-8080-exec-4] [4568423c] |<--OrderService.orderItem() time=1001ms
[nio-8080-exec-4] [4568423c] OrderController.request() time=1001ms
```

로그를 직접 분리해서 확인해보면 각각의 쓰레드 `nio-8080-exec-3`, `nio-8080-exec-4` 별로 로그가 정확하게 나누어 진 것을 확인할 수 있다.

쓰레드 로컬 - 주의사항

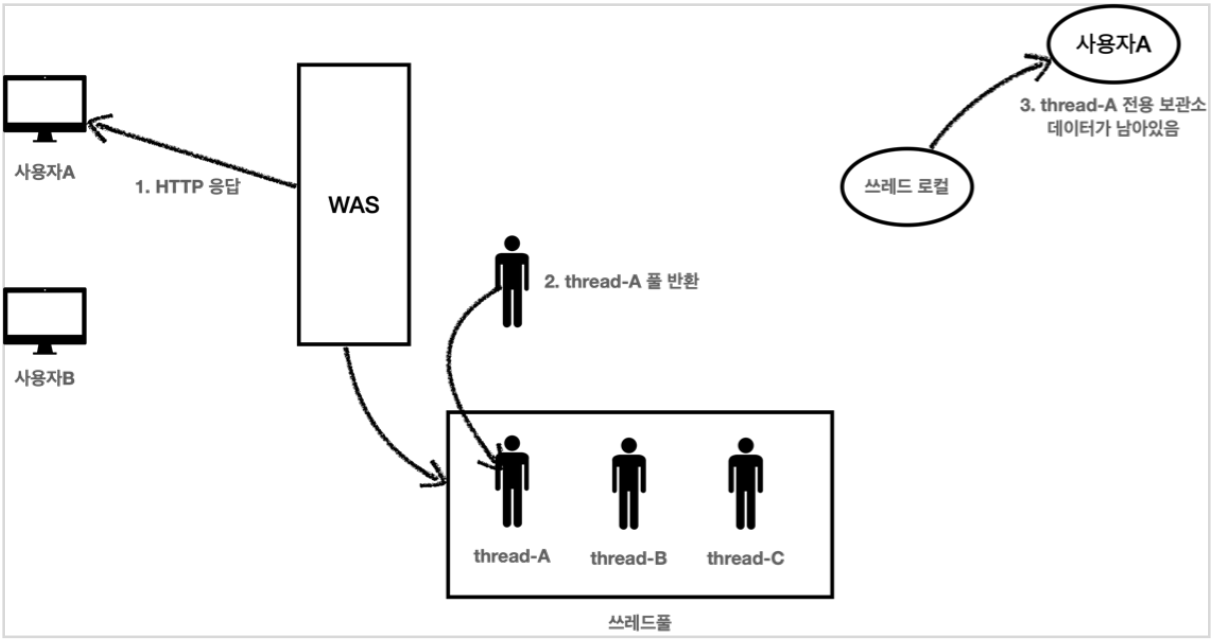
쓰레드 로컬의 값을 사용 후 제거하지 않고 그냥 두면 WAS(톰캣)처럼 쓰레드 풀을 사용하는 경우에 심각한 문제가 발생할 수 있다.
다음 예시를 통해서 알아보자.

사용자A 저장 요청



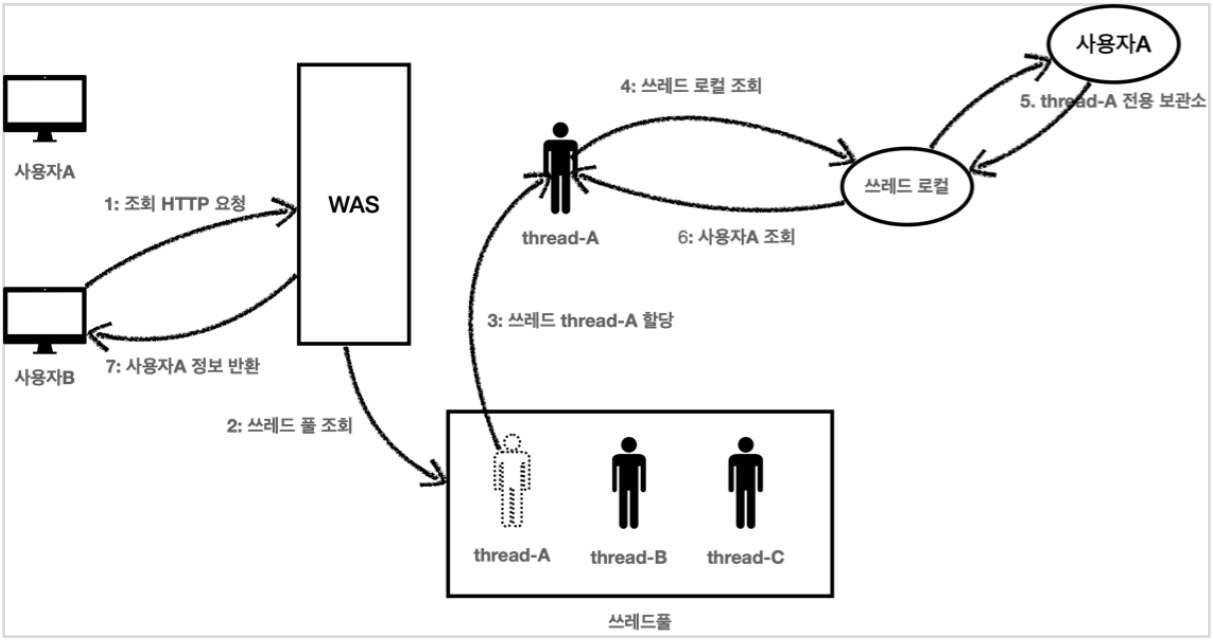
- 1. 사용자A가 저장 HTTP를 요청했다.
- 2. WAS는 쓰레드 풀에서 쓰레드를 하나 조회한다.
- 3. 쓰레드 `thread-A`가 할당되었다.
- 4. `thread-A`는 `사용자A`의 데이터를 쓰레드 로컬에 저장한다.
- 5. 쓰레드 로컬의 `thread-A` 전용 보관소에 `사용자A` 데이터를 보관한다.

사용자A 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.

사용자B 조회 요청



- 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.

결과적으로 사용자B는 사용자A의 데이터를 확인하게 되는 심각한 문제가 발생하게 된다.
이런 문제를 예방하려면 사용자A의 요청이 끝날 때 쓰레드 로컬의 값을 `ThreadLocal.remove()`를 통해서 꼭 제거해야 한다.
쓰레드 로컬을 사용할 때는 이 부분을 꼭! 기억하자.

정리