11. 동시성 컬렉션

#1.인강/0.자바/5.자바-고급1편

동시성 컬렉션이 필요한 이유1 - 시작

`java.util` 패키지에 소속되어 있는 컬렉션 프레임워크는 원자적인 연산을 제공할까?
예를 들어서 하나의 `ArrayList` 인스턴스에 여러 스레드가 동시에 접근해도 괜찮을까?
참고로 여러 스레드가 동시에 접근해도 괜찮은 경우를 **스레드 세이프(Thread Safe)**하다고 한다.
그렇다면 `ArrayList`는 스레드 세이프 할까?

다음 예제 코드를 보자.

```
package thread.collection.simple;

import java.util.ArrayList;
import java.util.List;

public class SimpleListMainV0 {

    public static void main(String[] args) {
        List<String> list = new ArrayList<>();

        // 스레드1, 스레드2가 동시에 실행 가정
        list.add("A"); // 스레드1 실행 가정
        list.add("B"); // 스레드2 실행 가정
        System.out.println(list);
    }
}
```

실행 결과

```
[A, B]
```

여기서는 멀티스레드를 사용하지 않지만, **스레드1과 스레드2가 동시에 다음 코드를 실행한다고 가정**해보자.

- 스레드1: `list`에 `A`를 추가한다.
- 스레드2: `list`에 `B`를 추가한다.

컬렉션에 데이터를 추가하는 `add()` 메서드를 생각해보면, 단순히 컬렉션에 데이터를 하나 추가하는 것뿐이다. 따라서 이것은 마치 연산이 하나만 있는 원자적인 연산처럼 느껴진다. 원자적인 연산은 쪼갤 수 없기 때문에 멀티스레드 상황에 문제가 되지 않는다.
물론 멀티스레드는 중간에 스레드의 실행 순서가 변경될 수 있으므로 `[A, B]` 또는, `[B, A]`로 데이터의 저장 순서는 변경될 수 있지만, 결과적으로 데이터는 모두 안전하게 저장될 것 같다.

하지만 컬렉션 프레임워크가 제공하는 대부분의 연산은 원자적인 연산이 아니다.

컬렉션 직접 만들기

이 부분을 이해하기 위해 아주 간단한 컬렉션을 직접 만들어보자.

```
package thread.collection.simple.list;

public interface SimpleList {
    int size();

    void add(Object e);

    Object get(int index);
}
```

- 직접 만들 컬렉션의 인터페이스이다.
- 크기 조회, 데이터 추가, 데이터 조회의 3가지 메서드만 가진다.

```
package thread.collection.simple.list;

import java.util.Arrays;
```

```
import static util.ThreadUtils.sleep;

public class BasicList implements SimpleList {

    private static final int DEFAULT_CAPACITY = 5;

    private Object[] elementData;
    private int size = 0;

    public BasicList() {
        elementData = new Object[DEFAULT_CAPACITY];
    }

    @Override
    public int size() {
        return size;
    }

    @Override
    public void add(Object e) {
        elementData[size] = e;
        sleep(100); // 멀티스레드 문제를 쉽게 확인하는 코드
        size++;
    }

    @Override
    public Object get(int index) {
        return elementData[index];
    }

    @Override
    public String toString() {
        return Arrays.toString(Arrays.copyOf(elementData, size)) + " size=" +
size + ", capacity=" + elementData.length;
    }

}
```

- 가장 간단한 컬렉션의 구현이다. 내부에서는 배열을 사용해서 데이터를 보관한다.
- `ArrayList`의 최소 구현 버전이라 생각하면 된다.
- `DEFAULT_CAPACITY`: 최대 5의 데이터를 저장할 수 있다.
- `size`: 저장한 데이터의 크기를 나타낸다.

- `add()` : 컬렉션에 데이터를 추가한다.
  - `sleep(100)` : 잠시 대기한다. 이렇게 하면 멀티스레드 상황에 발생하는 문제를 확인하기 쉽다.

만든 컬렉션을 실행해보자.

```
package thread.collection.simple;

import thread.collection.simple.list.BasicList;
import thread.collection.simple.list.SimpleList;

public class SimpleListMainV1 {

    public static void main(String[] args) {
        SimpleList list = new BasicList();
        list.add("A");
        list.add("B");
        System.out.println("list = " + list);
    }
}
```

실행 결과

```
list = [A, B] size=2, capacity=5
```

단일 스레드로 실행했기 때문에 아직까지는 아무런 문제 없이 잘 작동한다.

동시성 컬렉션이 필요한 이유2 - 동시성 문제

멀티스레드 문제 확인

add() - 원자적이지 않은 연산

```
public void add(Object e) {
    elementData[size] = e;
    sleep(100); // 멀티스레드 문제를 쉽게 확인하는 코드
    size++;
}
```

이 메서드는 단순히 데이터 하나를 추가하는 기능을 제공한다. 따라서 밖에서 보면 원자적인 것처럼 보인다.
이 메서드는 단순히 데이터를 추가하는 것으로 끝나지 않는다. 내부에 있는 배열에 데이터를 추가해야 하고, `size`도 함께 하나 증가시켜야 한다. 심지어 `size++` 연산 자체도 원자적이지 않다. `size++` 연산은 `size = size + 1` 연산과 같다.

이렇게 원자적이지 않은 연산을 멀티스레드 상황에 안전하게 사용하려면 `synchronized`, `Lock` 등을 사용해서 동기화를 해야한다.

이번에는 멀티스레드를 사용해서 실제 어떤 문제가 발생하는지 확인해보자.

```
package thread.collection.simple;

import thread.collection.simple.list.BasicList;
import thread.collection.simple.list.SimpleList;

import static util.MyLogger.log;

public class SimpleListMainV2 {

    public static void main(String[] args) throws InterruptedException {
        test(new BasicList());
    }

    private static void test(SimpleList list) throws InterruptedException {
        log(list.getClass().getSimpleName());

        // A를 리스트에 저장하는 코드
        Runnable addA = new Runnable() {
            @Override
            public void run() {
                list.add("A");
                log("Thread-1: list.add(A)");
            }
        };

        // B를 리스트에 저장하는 코드
        Runnable addB = new Runnable() {
            @Override
            public void run() {
                list.add("B");
```

```
                log("Thread-2: list.add(B)");
            }
        };

        Thread thread1 = new Thread(addA, "Thread-1");
        Thread thread2 = new Thread(addB, "Thread-2");
        thread1.start();
        thread2.start();
        thread1.join();
        thread2.join();
        log(list);
    }
}
```

실행 결과

```
09:48:13.989 [     main] BasicList
09:48:14.093 [ Thread-1] Thread-1: list.add(A)
09:48:14.096 [ Thread-2] Thread-2: list.add(B)
09:48:14.096 [     main] [B, null] size=2, capacity=5
```

- 참고로 어떤 스레드가 먼저 실행되는가에 따라 `[A, null]`이 결과로 나올 수도 있다.

실행 결과를 보면 `size`는 2인데, 데이터는 B 하나만 입력되어 있다! 어떻게 된 것일까?

과정1

```
public void add(Object e) {
    elementData[size] = e; // 스레드1, 스레드2 동시에 실행
    sleep(100);
    size++;
}
```

스레드1, 스레드2가 `elementData[size] = e` 코드를 동시에 수행한다. 여기서는 스레드1이 약간 빠르게 수행했다.

- 스레드1 수행: `elementData[0] = A`, `elementData[0]`의 값은 A가 된다.
- 스레드2 수행: `elementData[0] = B`, `elementData[0]`의 값은 A➜ B가 된다.

결과적으로 `elementData[0]`의 값은 B가 된다.

과정2

```
public void add(Object e) {
    elementData[size] = e;
    sleep(100); // 스레드1, 스레드2 동시에 실행
    size++;
}
```

스레드1, 스레드2가 `sleep()`에서 잠시 대기한다. 여기서 `sleep()`을 사용한 이유는 동시성 문제를 쉽게 확인하기 위해서다.
이 코드를 제거하면 `size++`이 너무 빨리 호출되기 때문에, 스레드1이 `add()` 메서드를 완전히 수행하고 나서 스레드2가 `add()` 메서드를 수행할 가능성이 높다.

당연한 이야기지만 `sleep()` 코드를 제거해도 멀티스레드 동시성 문제는 여전히 발생할 수 있다. (확률의 차이이다.)
예를 들어서 `sleep()` 코드를 제거해도 다음과 같은 상황이 발생할 수 있다.

```
public void add(Object e) {
    elementData[size] = e;
    size++; // 스레드1, 스레드2 size++ 실행 전 대기
}
```

과정3

```
public void add(Object e) {
    elementData[size] = e;
    sleep(100);
    size++; //스레드1, 스레드2 동시에 실행
}
```

여기서는 2가지 상황이 발생할 수 있다.

상황1

스레드1, 스레드2가 `size++` 코드를 동시에 수행한다. 여기서는 스레드1이 약간 빠르게 수행했다.

- 스레드1 수행: `size++`, `size`의 값은 1이 된다.
- 스레드2 수행: `size++`, `size`의 값은 1 ➜ 2가 된다.

결과적으로 `size`의 값은 2이 된다.

상황2

스레드1, 스레드2가 `size++` 코드를 동시에 수행한다. 여기서는 스레드1, 스레드2가 거의 동시에 실행되었다.

- 스레드1 수행: `size = size + 1` 연산이다. `size`의 값을 읽는다. 0이다.
- 스레드2 수행: `size = size + 1` 연산이다. `size`의 값을 읽는다. 0이다.

- 스레드1 수행: `size = 0 + 1` 연산을 수행한다.
- 스레드2 수행: `size = 0 + 1` 연산을 수행한다.

- 스레드1 수행: `size = 1` 대입을 수행한다.
- 스레드2 수행: `size = 1` 대입을 수행한다.

결과적으로 `size`의 값은 1이 된다.

우리가 본 케이스는 상황1이지만, `size++` 연산도 원자적인 연산이 아니므로 때때로 상황2가 될 수도 있다.
(따라서 로그에서 `size`값이 1로 출력될 가능성도 있다.)

컬렉션 프레임워크 대부분은 스레드 세이프 하지 않다.

우리가 일반적으로 자주 사용하는 `ArrayList`, `LinkedList`, `HashSet`, `HashMap` 등 수 많은 자료 구조들은 단순한 연산을 제공하는 것 처럼 보인다. 예를 들어서 데이터를 추가하는 `add()`와 같은 연산은 마치 원자적인 연산처럼 느껴진다. 하지만 그 내부에서는 수 많은 연산들이 함께 사용된다. 배열에 데이터를 추가하고, 사이즈를 변경하고, 배열을 새로 만들어서 배열의 크기도 늘리고, 노드를 만들어서 링크에 연결하는 등 수 많은 복잡한 연산이 함께 사용된다.

따라서 일반적인 컬렉션들은 절대로! 스레드 세이프 하지 않다!
단일 스레드가 컬렉션에 접근하는 경우라면 아무런 문제가 없지만, 멀티스레드 상황에서 여러 스레드가 동시에 컬렉션에 접근하는 경우라면 `java.util` 패키지가 제공하는 일반적인 컬렉션들은 사용하면 안된다! (물론 일부 예외도 있다. 뒤에서 설명한다.)
최악의 경우 실무에서 두 명의 사용자가 동시에 컬렉션에 데이터를 보관했는데, 코드에 아무런 문제가 없어 보이는데, 한명의 사용자 데이터가 사라질 수 있다.
그럼 어떻게 해야할까?

동시성 컬렉션이 필요한 이유3 - 동기화

컬렉션은 수 많은 복잡한 연산으로 이루어져 있다.
따라서 여러 스레드가 접근해야 한다면 `synchronized`, `Lock`등을 통해 안전한 임계 영역을 적절히 만들면 문제를 해결할 수 있다.

```java
package thread.collection.simple.list;

import java.util.Arrays;

import static util.ThreadUtils.sleep;

public class SyncList implements SimpleList {

    private static final int DEFAULT_CAPACITY = 5;

    private Object[] elementData;
    private int size = 0;

    public SyncList() {
        elementData = new Object[DEFAULT_CAPACITY];
    }

    @Override
    public synchronized int size() {
        return size;
    }

    @Override
    public synchronized void add(Object e) {
        elementData[size] = e;
        sleep(100); //멀티스레드 문제를 만드는 코드
        size++;
    }

    @Override
    public synchronized Object get(int index) {
        return elementData[index];
    }

    @Override
    public synchronized String toString() {
        return Arrays.toString(Arrays.copyOf(elementData, size)) + " size=" +
size + ", capacity=" + elementData.length;
    }

}
```

- 앞서 만든 `BasicList`에 `synchronized` 키워드만 추가했다.
- 모든 메서드가 동기화 되어 있으므로 멀티스레드 상황에 안전하게 사용할 수 있다.

SimpleListMainV2 - 코드 변경

```
public class SimpleListMainV2 {

    public static void main(String[] args) throws InterruptedException {
        //test(new BasicList());
        test(new SyncList());
    }
    ...
}
```

- `BasicList`를 사용하는 코드는 주석 처리하자.
- `SyncList`를 사용하도록 코드를 추가하자.

실행 결과

```
10:10:10.008 [     main] SyncList
10:10:10.115 [ Thread-1] Thread-1: list.add(A)
10:10:10.216 [ Thread-2] Thread-2: list.add(B)
10:10:10.216 [     main] [A, B] size=2, capacity=5
```

실행 결과를 보면 데이터가 `[A, B]`, `size=2`로 정상 수행된 것을 확인할 수 있다.

`add()` 메서드에 `synchronized`를 통해 안전한 임계 영역을 만들었기 때문에, 한 번에 하나의 스레드만 `add()` 메서드를 수행한다.

실행 순서

스레드1, 스레드2가 `add()` 코드를 동시에 수행한다. 여기서는 스레드1이 약간 빠르게 수행했다.

- 스레드1 수행: `add("A")`를 수행한다.
  - 락을 획득한다.
  - `size` 값은 0이다.
  - `elementData[0] = A`: `elementData[0]`의 값은 A가 된다.
  - `size++`을 호출해서 `size`는 1이 된다.
  - 락을 반납한다.
- 스레드2 수행: `add("B")`를 수행한다.
  - 스레드1이 락이 가져간 락을 획득하기 위해 `BLOCKED` 상태로 대기한다.

- 스레드 1이 락을 반납하면 락을 획득한다.
- `size` 값은 1이다.
- `elementData[1] = B`, `elementData[1]`의 값은 B가 된다. `size++`을 호출해서 `size`는 2이 된다.
- 락을 반납한다.

문제

`BasicList` 코드가 있는데, 이 코드를 거의 그대로 복사해서 `synchronized` 기능만 추가한 `SyncList`를 만들었다.
하지만 이렇게 되면 모든 컬렉션을 다 복사해서 동기화 용으로 새로 구현해야 한다. 이것은 매우 비효율적이다.

동시성 컬렉션이 필요한 이유4 - 프록시 도입

`ArrayList`, `LinkedList`, `HashSet`, `HashMap`등의 코드도 모두 복사해서 `synchronized` 기능을 추가한 코드를 만들어야 할까? 예를 들어서 다음과 같이 말이다.

- `ArrayList` ➜ `SyncArrayList`
- `LinkedList` ➜ `SyncLinkedList`

하지만 이렇게 코드를 복사해서 만들면 이후에 구현이 변경될 때 같은 모양의 코드를 2곳에서 변경해야 한다.
기존 코드를 그대로 사용하면서 `synchronized` 기능만 살짝 추가하고 싶다면 어떻게 하면 좋을까?
예를 들어서 `BasicList`는 그대로 사용하면서, 멀티스레드 상황에 동기화가 필요할 때만 `synchronized` 기능을 살짝 추가하고 싶다면 어떻게 하면 될까?

이럴때 사용하는 것이 바로 프록시이다.

프록시(Proxy)

우리말로 대리자, 대신 처리해주는 자라는 뜻이다.
프록시를 쉽게 풀어서 설명하자면 친구에게 대신 음식을 주문해달라고 부탁하는 상황을 생각해 볼 수 있다. 예를 들어, 당신이 피자를 먹고 싶은데, 직접 전화하는 게 부담스러워서 친구에게 대신 전화해서 피자를 주문해달라고 부탁한다고 해보자. 친구가 피자 가게에 전화를 걸어 주문하고, 피자가 도착하면 당신에게 가져다주는 것이다. 여기서 친구가 프록시 역할을 하는 것이다.

- 나(클라이언트) ➜ 피자 가게(서버)
- 나(클라이언트) ➜ 친구(프록시) ➜ 피자 가게(서버)

객체 세상에도 이런 프록시를 만들 수 있다. 여기서는 프록시가 대신 동기화(`synchronized`) 기능을 처리해주는 것이다.
코드를 만들면서 확인해보자.

```
package thread.collection.simple.list;

public class SyncProxyList implements SimpleList {

    private SimpleList target;

    public SyncProxyList(SimpleList target) {
        this.target = target;
    }

    @Override
    public synchronized void add(Object e) {
        target.add(e);
    }

    @Override
    public synchronized Object get(int index) {
        return target.get(index);
    }

    @Override
    public synchronized int size() {
        return target.size();
    }

    @Override
    public synchronized String toString() {
        return target.toString() + " by " + this.getClass().getSimpleName();
    }
}
```

- 프록시 역할을 하는 클래스이다.
- `SyncProxyList`는 `BasicList`와 같은 `SimpleList` 인터페이스를 구현한다.
- 이 클래스는 생성자를 통해 `SimpleList target`을 주입 받는다. 여기에 실제 호출되는 대상이 들어간다.
- 이 클래스는 마치 빈껍데기 처럼 보인다. 이 클래스의 역할은 모든 메서드에 `synchronized`를 걸어주는 일 뿐이다. 그리고나서 `target`에 있는 같은 기능을 호출한다.

- 이 프록시 클래스는 `synchronized`만 걸고, 그 다음에 바로 실제 호출해야 하는 원본 대상(`target`)을 호출한다.

SimpleListMainV2 - 코드 변경

```
public class SimpleListMainV2 {

    public static void main(String[] args) throws InterruptedException {
        //test(new BasicList());
        //test(new SyncList());
        test(new SyncProxyList(new BasicList()));
    }
    ...
}
```

- `SyncProxyList`는 프록시이다. 생성자에 실제 대상인 `BasicList`가 필요하다.

당연한 이야기지만 다음과 같이 나누어 작성해도 된다.

```
SimpleList basicList = new BasicList();
SimpleList proxyList = new SyncProxyList(basicList);
test(list)
```

기존에 `BasicList`를 직접 사용하고 있었다면, 이제 중간에 프록시를 사용하므로 다음과 같은 구조로 변경된다.

- **기존 구조**: 클라이언트 ➡ `BasicList`(서버)
- **변경 구조**: 클라이언트 ➡ `SyncProxyList`(프록시)➡ `BasicList`(서버)

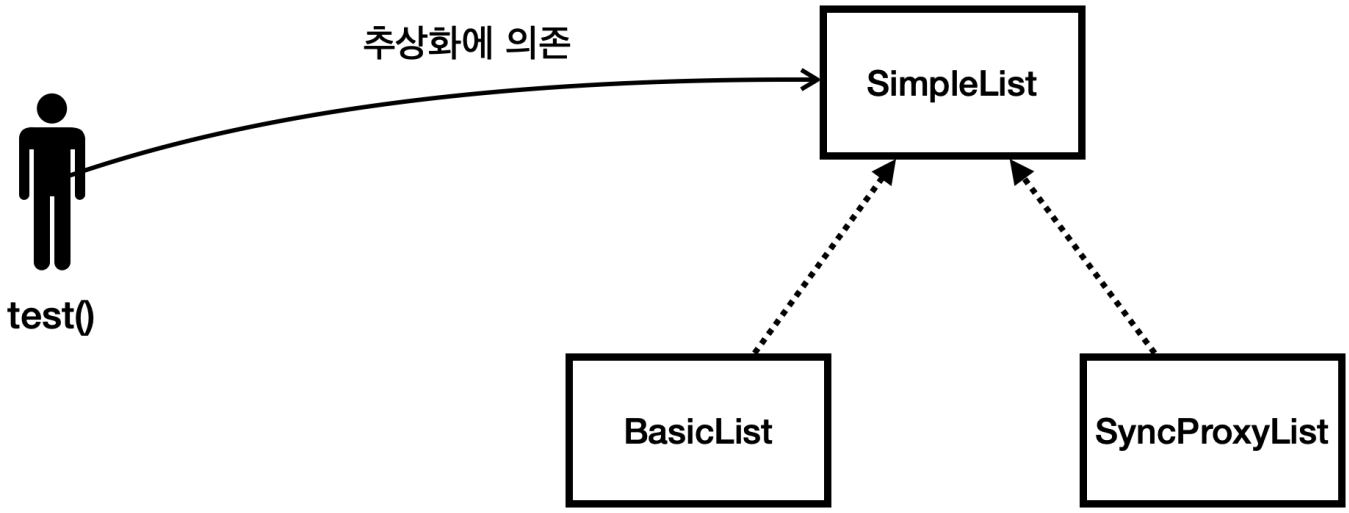
실행 결과

```
10:39:58.693 [     main] SyncProxyList
10:39:58.800 [ Thread-1] Thread-1: list.add(A)
10:39:58.905 [ Thread-2] Thread-2: list.add(B)
10:39:58.905 [     main] [A, B] size=2, capacity=5 by SyncProxyList
```

실행 결과를 보면 `[A, B]`, `size=2`로 `synchronized`를 통한 동기화가 잘 이루어진 것을 확인할 수 있다.
프록시를 사용한 구조를 분석해보자.

프록시 구조 분석

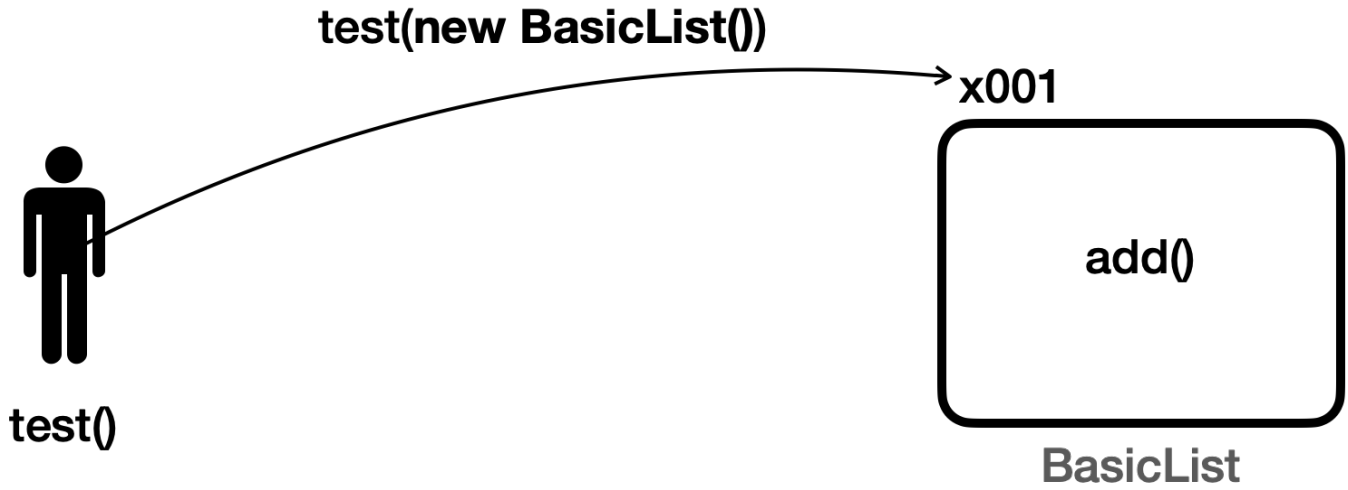
정적 의존 관계



- 그림과 같이 정적인 클래스의 의존 관계를 정적 의존 관계라 한다.
- `test()` 메서드를 클라이언트라고 가정하자. `test()` 메서드는 `SimpleList`라는 인터페이스에만 의존한다.
  - 이것을 추상화에 의존한다고 표현한다.
- 덕분에 `SimpleList` 인터페이스의 구현체인 `BasicList`, `SyncList`, `SyncProxyList` 중에 어떤 것을 사용하든, 클라이언트인 `test()`의 코드는 전혀 변경하지 않아도 된다.
- 클라이언트인 `test()` 입장에서 생각해보면 `BasicList`가 넘어올지, `SyncProxyList`가 넘어올지 알 수 없다. 단순히 `SimpleList`의 구현체 중의 하나가 넘어와서 실행된다는 정도만 알 수 있다. 그래서 클라이언트인 `test()`는 매우 유연하다. `SimpleList`의 어떤 구현체든지 다 받아들일 수 있다.
  - `test(SimpleList list){...}`

런타임 의존 관계 - BasicList

먼저 `BasicList`를 사용하는 예를 보자.

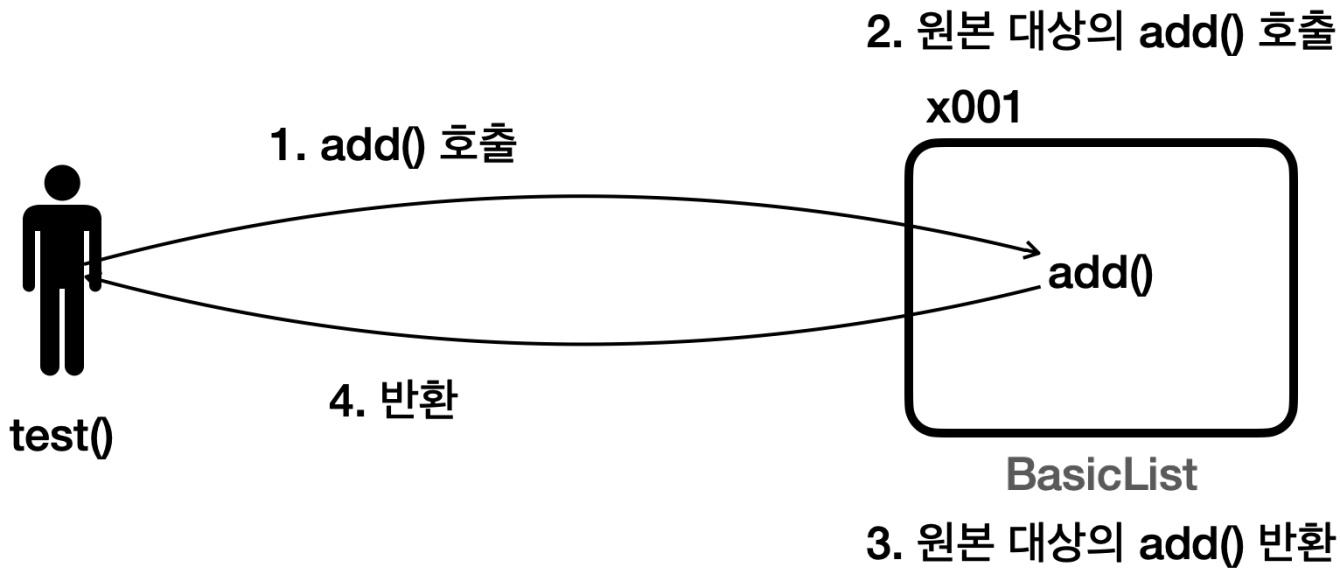


그림과 같이 실제 런타임에 발생하는 인스턴스의 의존 관계를 런타임 의존 관계라 한다.

먼저 간단한 `BasicList`를 직접 사용하는 경우부터 알아보자.

- `test(new BasicList())`를 실행하면 `BasicList(x001)`의 인스턴스가 만들어지면서 `test()` 메서드에 전달된다.
- `test()` 메서드는 `BasicList(x001)` 인스턴스의 참조를 알고 사용하게 된다.
  - `test(SimpleList list=x001)`

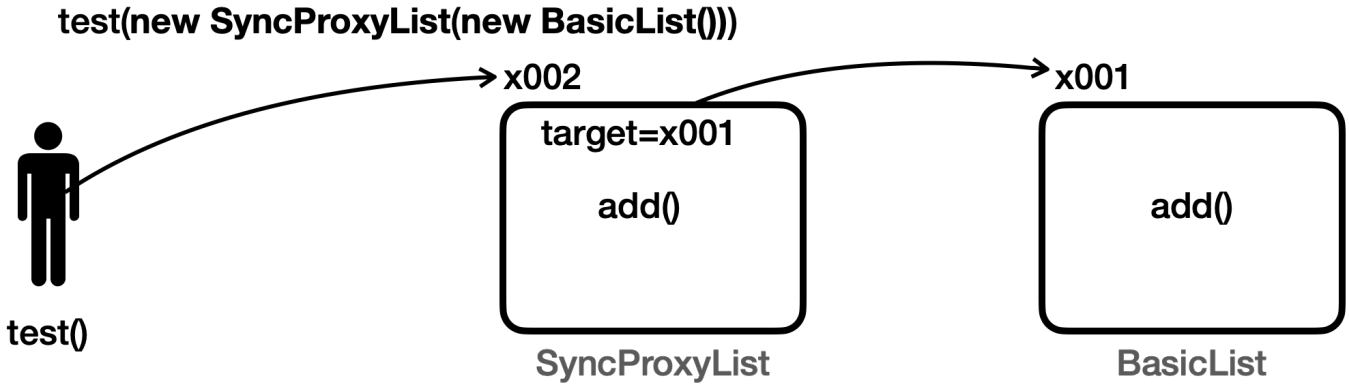
BasicList - add() 호출 과정



- `test()` 메서드에서 스레드를 만들고, 스레드에 있는 `run()`에서 `list.add()`를 호출한다.
- 그림은 간단하게 `test()`에서 호출하는 것으로 표현하겠다.
- `BasicList(x001)` 인스턴스에 있는 `add()`가 호출된다.

런타임 의존 관계 - SyncProxyList

이번에는 `BasicList`가 아니라 `SyncProxyList`를 사용하는 예를 보자.



다음 두 코드는 같은 코드이므로 쉽게 풀어서 설명하겠다.

```
test(new SyncProxyList(new BasicList()))
```

```
SimpleList basicList = new BasicList();
SimpleList list = new SyncProxyList(basicList);
test(list)
```

`SimpleList basicList = new BasicList()`

- 먼저 `BasicList(x001)`의 인스턴스가 만들어진다.

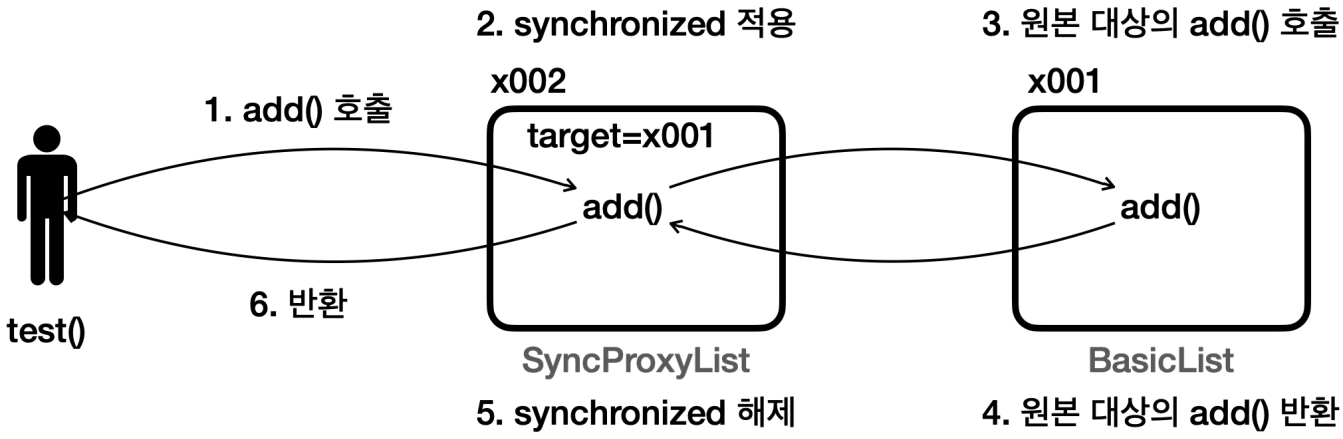
`SimpleList list = new SyncProxyList(basicList)`

- 앞서 만든 `BasicList(x001)`의 참조를 `SyncProxyList`의 생성자에 전달하며 `SyncProxyList(x002)`가 만들어진다.
- 내부에는 원본 대상을 가르키는 `target` 변수를 포함하고 있다. 이 변수는 `BasicList(x001)`의 참조를 보관한다.

`test(list)`

- `test()` 메서드는 `SyncProxyList(x002)` 인스턴스를 사용하게 된다.
- `test(SimpleList list=x002)`

SyncProxyList - add() 호출 과정



1. `test()` 메서드에서 스레드를 만들고, 스레드에 있는 `run()`에서 `list.add()`를 호출한다.
    - `SyncProxyList(x002)`에 있는 `add()`가 호출된다.
    - 그림은 간단하게 `test()`에서 호출하는 것으로 표현하겠다.
2. 프록시인 `SyncProxyList`는 `synchronized`를 적용한다. 그리고 나서 `target`에 있는 `add()`를 호출한다.
3. 원본 대상인 `BasicList(x001)`의 `add()`가 호출된다.

4. 원본 대상의 호출이 끝나면 결과를 반환한다.
5. `SyncProxyList`에 있는 `add()`로 흐름이 돌아온다. 메서드를 반환하면서 `synchronized`를 해제한다.
6. `test()`로 흐름이 돌아온다.

프록시 정리

- 프록시인 `SyncProxyList`는 원본인 `BasicList`와 똑같은 `SimpleList`를 구현한다. 따라서 클라이언트인 `test()` 입장에서는 원본 구현체가 전달되든, 아니면 프록시 구현체가 전달되든 아무런 상관이 없다. 단지 수많은 `SimpleList`의 구현체 중의 하나가 전달되었다고 생각할 뿐이다.
- 클라이언트 입장에서 보면 프록시는 원본과 똑같이 생겼고, 호출할 메서드도 똑같다. 단지 `SimpleList`의 구현체일 뿐이다.
- 프록시는 내부에 원본을 가지고 있다. 그래서 프록시가 필요한 일부의 일을 처리하고, 그다음에 원본을 호출하는 구조를 만들 수 있다. 여기서 프록시는 `synchronized`를 통한 동기화를 적용한다.
- 프록시가 동기화를 적용하고 원본을 호출하기 때문에 원본 코드도 이미 동기화가 적용된 상태로 호출된다.

여기서 중요한 핵심은 원본 코드인 `BasicList`를 전혀 손대지 않고, 프록시인 `SyncProxyList`를 통해 동기화 기능을 적용했다는 점이다.
또한 이후에 `SimpleList`를 구현한 `BasicLinkedList` 같은 연결 리스트를 만들더라도 서로 같은 인터페이스를 사용하기 때문에 `SyncProxyList`를 그대로 활용할 수 있다.
쉽게 이야기해서 `SyncProxyList` 프록시 하나로 `SimpleList` 인터페이스의 모든 구현체를 동기화 할 수 있다.

프록시 패턴

지금까지 우리가 구현한 것이 바로 프록시 패턴이다.
프록시 패턴(Proxy Pattern)은 객체지향 디자인 패턴 중 하나로, 어떤 객체에 대한 접근을 제어하기 위해 그 객체의 대리인 또는 인터페이스 역할을 하는 객체를 제공하는 패턴이다. 프록시 객체는 실제 객체에 대한 참조를 유지하면서, 그 객체에 접근하거나 행동을 수행하기 전에 추가적인 처리를 할 수 있도록 한다.

프록시 패턴의 주요 목적

- **접근 제어**: 실제 객체에 대한 접근을 제한하거나 통제할 수 있다.
- **성능 향상**: 실제 객체의 생성을 지연시키거나 캐싱하여 성능을 최적화할 수 있다.
- **부가 기능 제공**: 실제 객체에 추가적인 기능(로깅, 인증, 동기화 등)을 투명하게 제공할 수 있다.

> **참고**: 실무에서 프록시 패턴은 자주 사용된다. 스프링의 AOP 기능은 사실 이런 프록시 패턴을 극한으로 적용하는 예이다. 참고로 **스프링 핵심 원리 - 고급편**에서 이 부분을 자세히 다룬다.

자바 동시성 컬렉션1 - synchronized

자바가 제공하는 `java.util` 패키지에 있는 컬렉션 프레임워크들은 대부분 스레드 안전(Thread Safe)하지 않다.

우리가 일반적으로 사용하는 `ArrayList`, `LinkedList`, `HashSet`, `HashMap` 등 수 많은 기본 자료 구조들은 내부에서 수 많은 연산들이 함께 사용된다. 배열에 데이터를 추가하고 사이즈를 변경하고, 배열을 새로 만들어서 배열의 크기도 늘리고, 노드를 만들어서 링크에 연결하는 등 수 많은 복잡한 연산이 함께 사용된다.

그렇다면 처음부터 모든 자료 구조에 `synchronized`를 사용해서 동기화를 해두면 어떨까?
`synchronized`, `Lock`, `CAS` 등 모든 방식은 정도의 차이는 있지만 성능과 트레이드 오프가 있다.
결국 **동기화를 사용하지 않는 것이 가장 빠르다.**
그리고 컬렉션이 항상 멀티스레드에서 사용되는 것도 아니다. 미리 동기화를 해둔다면 단일 스레드에서 사용할 때 동기화로 인해 성능이 저하된다. 따라서 동기화의 필요성을 정확히 판단하고 꼭 필요한 경우에만 동기화를 적용하는 것이 필요하다.

> **참고**: 과거에 자바는 이런 실수를 한번 했다. 그것이 바로 `java.util.Vector` 클래스이다. 이 클래스는 지금의 `ArrayList`와 같은 기능을 제공하는데, 메서드에 `synchronized`를 통한 동기화가 되어 있다. 쉽게 이야기해서 동기화된 `ArrayList`이다. 그러나 이에 따라 단일 스레드 환경에서도 불필요한 동기화로 성능이 저하되었고, 결과적으로 `Vector`는 널리 사용되지 않게 되었다. 지금은 하위 호환을 위해서 남겨져 있고 다른 대안이 많기 때문에 사용을 권장하지 않는다.

좋은 대안으로는 우리가 앞서 배운 것처럼 `synchronized`를 대신 적용해 주는 프록시를 만드는 방법이 있다.
`List`, `Set`, `Map` 등 주요 인터페이스를 구현해서 `synchronized`를 적용할 수 있는 프록시를 만들면 된다.
이 방법을 사용하면 기존 코드를 그대로 유지하면서 필요한 경우에만 동기화를 적용할 수 있다.

자바는 컬렉션을 위한 프록시 기능을 제공한다.

자바 synchronized 프록시

```
package thread.collection.java;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
```

```
public class SynchronizedListMain {

    public static void main(String[] args) {
        List<String> list = Collections.synchronizedList(new ArrayList<>());
        list.add("data1");
        list.add("data2");
        list.add("data3");
        System.out.println(list.getClass());
        System.out.println("list = " + list);
    }
}
```

실행 결과

```
class java.util.Collections$SynchronizedRandomAccessList
list = [data1, data2, data3]
```

Collections.synchronizedList(target)

이 코드는 다음과 같다.

```
public static <T> List<T> synchronizedList(List<T> list) {
    return new SynchronizedRandomAccessList<>(list);
}
```

- 참고로 코드에서 핵심이 되는 부분만 추려서 보여주었다.

이 코드는 결과적으로 다음과 같은 코드이다.

```
new SynchronizedRandomAccessList<>(new ArrayList())
```

`SynchronizedRandomAccessList`는 `synchronized`를 추가하는 프록시 역할을 한다.

- 클라이언트 ➜ `ArrayList`
- 클라이언트 ➜ `SynchronizedRandomAccessList`(프록시) ➜ `ArrayList`

예를 들어서 이 클래스의 `add()` 메서드를 보면 `synchronized` 코드 블럭을 적용하고, 그 다음에 원본 대상의 `add()`를 호출하는 것을 확인할 수 있다.

```
public boolean add(E e) {
    synchronized (mutex) {
        return c.add(e);
    }
}
```

`Collections`는 다음과 같이 다양한 `synchronized` 동기화 메서드를 지원한다. 이 메서드를 사용하면 `List`, `Collection`, `Map`, `Set` 등 다양한 동기화 프록시를 만들어낼 수 있다.

- `synchronizedList()`
- `synchronizedCollection()`
- `synchronizedMap()`
- `synchronizedSet()`
- `synchronizedNavigableMap()`
- `synchronizedNavigableSet()`
- `synchronizedSortedMap()`
- `synchronizedSortedSet()`

`Collections`가 제공하는 동기화 프록시 기능 덕분에 스레드 안전하지 않은 수 많은 컬렉션들을 매우 편리하게 스레드 안전한 컬렉션으로 변경해서 사용할 수 있다.

synchronized 프록시 방식의 단점

하지만 `synchronized` 프록시를 사용하는 방식은 다음과 같은 단점이 있다.

- 첫째, 동기화 오버헤드가 발생한다. 비록 `synchronized` 키워드가 멀티스레드 환경에서 안전한 접근을 보장하지만, 각 메서드 호출 시마다 동기화 비용이 추가된다. 이로 인해 성능 저하가 발생할 수 있다.
- 둘째, 전체 컬렉션에 대해 동기화가 이루어지기 때문에, 잠금 범위가 넓어질 수 있다. 이는 잠금 경합(lock contention)을 증가시키고, 병렬 처리의 효율성을 저하시키는 요인이 된다. 모든 메서드에 대해 동기화를 적용하다 보면, 특정 스레드가 컬렉션을 사용하고 있을 때 다른 스레드들이 대기해야 하는 상황이 빈번해질 수 있다.
- 셋째, 정교한 동기화가 불가능하다. `synchronized` 프록시를 사용하면 컬렉션 전체에 대한 동기화가 이루어지지만, 특정 부분이나 메서드에 대해 선택적으로 동기화를 적용하는 것은 어렵다. 이는 과도한 동기화로 이어질 수 있다.

쉽게 이야기해서 이 방식은 단순 무식하게 모든 메서드에 `synchronized`를 걸어버리는 것이다. 따라서 동기화에 대한 최적화가 이루어지지 않는다. 자바는 이런 단점을 보완하기 위해 `java.util.concurrent` 패키지에 동시성 컬렉션(concurrent collection)을 제공한다.

자바 동시성 컬렉션2 - 동시성 컬렉션

동시성 컬렉션

자바 1.5부터 동시성에 대한 많은 혁신이 이루어졌다. 그 중에 동시성을 위한 컬렉션도 있다.

여기서 말하는 동시성 컬렉션은 스레드 안전한 컬렉션을 뜻한다.

`java.util.concurrent` 패키지에는 고성능 멀티스레드 환경을 지원하는 다양한 동시성 컬렉션 클래스들을 제공한다. 예를 들어, `ConcurrentHashMap`, `CopyOnWriteArrayList`, `BlockingQueue` 등이 있다. 이 컬렉션들은 더 정교한 잠금 메커니즘을 사용하여 동시 접근을 효율적으로 처리하며, 필요한 경우 일부 메서드에 대해서만 동기화를 적용하는 등 유연한 동기화 전략을 제공한다.

여기에 다양한 성능 최적화 기법들이 적용되어 있는데, `synchronized`, `Lock`(`ReentrantLock`), `CAS`, 분할 잠금 기술(segment lock)등 다양한 방법을 섞어서 매우 정교한 동기화를 구현하면서 동시에 성능도 최적화했다.

각각의 최적화는 매우 어렵게 구현되어 있기 때문에, 자세한 구현을 이해하는 것 보다는, 멀티스레드 환경에 필요한 동시성 컬렉션을 잘 선택해서 사용할 수 있으면 충분하다.

동시성 컬렉션의 종류

- `List`
  - `CopyOnWriteArrayList` ➜ `ArrayList`의 대안
- `Set`
  - `CopyOnWriteArraySet` ➜ `HashSet`의 대안
  - `ConcurrentSkipListSet` ➜ `TreeSet`의 대안(정렬된 순서 유지, `Comparator` 사용 가능)
- `Map`
  - `ConcurrentHashMap`: `HashMap`의 대안
  - `ConcurrentSkipListMap`: `TreeMap`의 대안(정렬된 순서 유지, `Comparator` 사용 가능)
- `Queue`
  - `ConcurrentLinkedQueue`: 동시성 큐, 비 차단(non-blocking) 큐이다.
- `Deque`
  - `ConcurrentLinkedDeque`: 동시성 데크, 비 차단(non-blocking) 큐이다.

참고로 `LinkedHashSet`, `LinkedHashMap`처럼 입력 순서를 유지하는 동시에 멀티스레드 환경에서 사용할 수 있는 `Set`, `Map` 구현체는 제공하지 않는다. 필요하다면 `Collections.synchronizedXxx()`를 사용해야 한다.

스레드를 차단하는 블로킹 큐도 알아보자.

- `BlockingQueue`
  - `ArrayBlockingQueue`
    - 크기가 고정된 블로킹 큐
    - 공정(fair) 모드를 사용할 수 있다. 공정(fair) 모드를 사용하면 성능이 저하될 수 있다.
  - `LinkedBlockingQueue`
    - 크기가 무한하거나 고정된 블로킹 큐
  - `PriorityBlockingQueue`
    - 우선순위가 높은 요소를 먼저 처리하는 블로킹 큐
  - `SynchronousQueue`
    - 데이터를 저장하지 않는 블로킹 큐로, 생산자가 데이터를 추가하면 소비자가 그 데이터를 받을 때까지 대기한다. 생산자-소비자 간의 직접적인 핸드오프(hand-off) 메커니즘을 제공한다. 쉽게 이야기해서 중간에 큐 없이 생산자, 소비자가 직접 거래한다.
  - `DelayQueue`
    - 지연된 요소를 처리하는 블로킹 큐로, 각 요소는 지정된 지연 시간이 지난 후에야 소비될 수 있다. 일정 시간이 지난 후 작업을 처리해야 하는 스케줄링 작업에 사용된다.

List - 예시

```
package thread.collection.java;

import java.util.List;
import java.util.concurrent.CopyOnWriteArrayList;

public class ListMain {

    public static void main(String[] args) {
        List<Integer> list = new CopyOnWriteArrayList<>();
        list.add(1);
        list.add(2);
        list.add(3);
        System.out.println("list = " + list);
    }
}
```

실행 결과

```
list = [1, 2, 3]
```

`CopyOnWriteArrayList`은 `ArrayList`의 대안이다.

Set - 예시

```
package thread.collection.java;

import java.util.Set;
import java.util.concurrent.ConcurrentSkipListSet;
import java.util.concurrent.CopyOnWriteArraySet;

public class SetMain {

    public static void main(String[] args) {
        Set<Integer> copySet = new CopyOnWriteArraySet<>();
        copySet.add(1);
        copySet.add(2);
        copySet.add(3);
        System.out.println("copySet = " + copySet);

        Set<Integer> skipSet = new ConcurrentSkipListSet<>();
        skipSet.add(3);
        skipSet.add(2);
        skipSet.add(1);
        System.out.println("skipSet = " + skipSet);
    }
}
```

실행 결과

```
copySet = [1, 2, 3]
skipSet = [1, 2, 3]
```

- `CopyOnWriteArraySet`은 `HashSet`의 대안이다.
- `ConcurrentSkipListSet`은 `TreeSet`의 대안이다. 데이터의 정렬 순서를 유지한다. `Comparator` 사용 가능

Map - 예시

```
package thread.collection.java;
```

```java
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ConcurrentSkipListMap;

public class MapMain {

    public static void main(String[] args) {
        Map<Integer, String> map1 = new ConcurrentHashMap<>();
        map1.put(3, "data3");
        map1.put(2, "data2");
        map1.put(1, "data1");
        System.out.println("map1 = " + map1);

        Map<Integer, String> map2 = new ConcurrentSkipListMap<>();
        map2.put(2, "data2");
        map2.put(3, "data3");
        map2.put(1, "data1");
        System.out.println("map2 = " + map2);

    }
}
```

실행 결과

```
map1 = {1=data1, 3=data3, 2=data2}
map2 = {1=data1, 2=data2, 3=data3}
```

- `ConcurrentHashMap`은 `HashMap`의 대안이다.
- `ConcurrentSkipListMap`은 `TreeMap`의 대안이다. 데이터의 정렬 순서를 유지한다. `Comparator` 사용 가능

정리

자바가 제공하는 동시성 컬렉션은 멀티스레드 상황에 최적의 성능을 낼 수 있도록 다양한 최적화 기법이 적용되어 있다. 따라서 `Collections.synchronizedXxx`를 사용하는 것 보다 더 좋은 성능을 제공한다.

당연한 이야기지만 동시성은 결국 성능과 트레이드 오프가 있다. 따라서 단일 스레드가 컬렉션을 사용하는 경우에는 동시성 컬렉션이 아닌 일반 컬렉션을 사용해야 한다.

반대로 멀티스레드 상황에서 일반 컬렉션을 사용하면 정말 해결하기 어려운 버그를 만날 수 있다. 세상에서 가장 해결하기 어려운 버그가 멀티스레드로 인해 발생한 버그이다. 이러한 이유로 멀티스레드 환경에서는 동시성 컬렉션을 적절히 활용해서 버그를 예방하고 성능을 최적화하는 것이 중요하다. 동시성 컬렉션을 사용하면 코드의 안정성과 효율성을 높일 수 있으며, 예상치 못한 동시성 문제도 방지할 수 있다.