동시성 컬렉션

Java.util의 컬렉션은 thread-safe한가?

add(Object e)는 원자적이지 않음 → 배열에 데이터추가, size증가(size++)

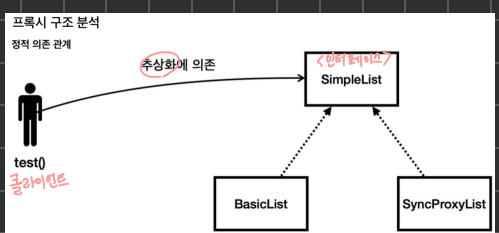
```
public void add(Object e) {
    elementData[size] = e; // 스레드1, 스레드2 동시에 실행
    sleep(100);
    size++;
}
```

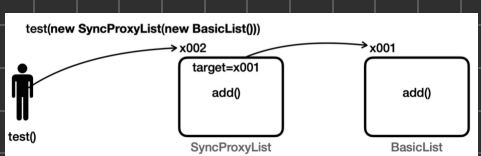
→ 스레드1이 추가한 값이 스레드2의 값에 덮어씌워지게됨

여러 스레드에서 동시에 컬렉션에 접근하는 경우라면 사용하지 말것(일부 예외존재)

프록시(Proxy)

: 기존 코드를 그대로 사용하면서 synchronized 기능을 추가하고 싶을때, 대신 처리해주는것





기존의 클래스를 생성자의 인수로 받아 synchronized 동기화처리. → test(new SyncProxy클래스(new Basic클래스()));

- Before : 클라이언트 → BasicList(서버)
- After : 클라이언트 → SyncProxy클래스(프록시) → Basic클래스(서버)

Proxy 정리

- test() 입장에서는 원본 구현체든지 프록시 구현체가 전달되든 상관X
- 프록시 내부에 원본을 갖고있어 프록시가 필요한 일을 처리하고, 그 다음에 원본 호출 & synchronized 동기화적용

※ 원본 클래스를 수정하지 않고 프록시를 통해 동기화 기능적용

Proxy 패턴

: 어떤 객체에 대한 접근을 제어하기 위해 그 객체의 대리인 혹은 인터페이스 역할을 하는 객체를 제공하는 패턴

프록시 객체는 실제 객체에 대한 참조를 유지하면서 그 객체에 접근하거나 해당을 수행하기 전 후 추가적인 처리 가능해짐

프록시 패턴의 목적

1. 접근 제어 : 실제 객체에 대한 접근 통제
2. 성능향상 : 실제 객체의 생성을 지연, 캐싱하여 성능 최적화
3. 부가기능제공 : 실제 객체에 추가적인 기능(로깅, 인증, 동기화)를 투명하게 제공 → Spring의 AOP

java.util.Collections 에서 제공하는 synchronized 프록시

ex) Collections.synchronizedList(new ArrayList<>());

↓ 내부

SynchronizedRandomAccessList : synchronized를 추가하는 프록시역할

- client → ArrayList
- client → SynchronizedRandomAccessList(proxy) → ArrayList

synchronized 프록시 방식의 단점

1. 동기화 오버헤드
2. 잠금 범위가 넓어져 lock contention(경합)을 증가시키고 병렬 처리 효율성 저하
   모든 메서드에 동기화 → 특정 쓰레드가 컬렉션을 사용하고 있다면 다른 쓰레드들은 대기
3. 정교한 동기화 불가능, 선택적 동기화가 어려움

↓ 단점 커버

java.util.concurrent 패키지의 동시성 컬렉션

ex) ConcurrentHashMap, CopyOnWriteArrayList, BlockingQueue 등

∨ 필요한 일부 메서드에 동기화 적용 가능

동시성 컬렉션의 종류

- List
  - CopyOnWriteArrayList → ArrayList의 대안
- Set
  - CopyOnWriteArraySet → HashSet의 대안
  - ConcurrentSkipListSet → TreeSet의 대안(정렬된 순서 유지, Comparator 사용 가능)
- Map
  - ConcurrentHashMap: HashMap의 대안
  - ConcurrentSkipListMap: TreeMap의 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.

- 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.

LinkedHashSet, LinkedHashMap - 입력순서 유지 필요 → 그래서 구체적 제공X
Collections의 동기화 사용