11장 동시성

분류	이펙티브 자바
태그	서적
시작일자	@2022년 6월 13일 오후 7:44
종료일자	@2022년 6월 21일 오후 7:04
참고	참고 블로그 [자바봄 기술 블로그] 참고 github.ido
스터디 링크	GitHub Link [이펙티브 자바 스터디]
서적	
Java	

▼ 목차

아이템 78. 공유 중인 가변 데이터는 동기화해 사용하라.

❓ **동기화를 꼭 배워야 하는건가?**

> 스레드는 여러 활동을 동시에 수행할 수 있게 해줍니다.
> 여러 활동을 동시에 처리한다면 더 빠르게 처리할 수 있지만 멀티 프로세서를 제대로 활용하려고 하면 동시성의 개념을 알아야합니다.

동기화란

> 프로세스나 스레드들이 서로 알고 있는 공유중인 데이터가 같은 것을 의미한다.
> 멀티 코어 프로세서의 힘을 제대로 활용하려면 멀티스레드 프로그래밍을 해야하며 여러 쓰레드가 동시에 접근하는 만큼 동기화에 중의를 기울여야 한다.

동기화는 2가지 기능을 제공한다.

1. **배타적 실행 : 현재 사용중인 스레드만 접근이 가능하고, 다른 스레드가 접근하지 못하게 한다.**

- 일관성이 깨진 상태를 볼 수 없게 한다. **즉 항상 일관성이 지켜진 상태로 있게 해준다.**

2. **스레드 사이 안정적인 통신 : 어떤 쓰레드가 변경한 데이터를 다른 쓰레드에서 읽을 수 있게 한다**
   - 변경된 데이터의 최종 결과값을 읽을 수 있게 한다는 의미이다.

❓ 자바 언어는 long과 double을 제외한 변수를 읽고 쓰는 동작이 원자적이다?

> 책에 명시된 내용에서 *"언어 명세상 long과 double 외의 변수를 읽고 쓰는 동작은 원자적 이다"* 라는 내용이 나옵니다.
> 읽고 쓰는 동작이 원자적(Atomic) 이라는 뜻은 여러 쓰레드가 같은 변수를 동기화없이 수정하는 중이라도
> **어떤 쓰레드가 정상적으로 저장한 값을 온전히 읽어온다는 것을 보장하는 의미입니다.**

❓ 그러면 원자적 데이터를 읽고 쓸 때는 동기화를 하지 않아도 되나?

> 정답은 X이다.
> 쓰레드가 필드를 읽을 때 하상 수정이 반영된 값을 얻는다고 보장하지만, *한 쓰레드가 저장한 값이 다른 쓰레드에게 보이는가* 는 보장하지 않습니다.
> 즉 다른 쓰레드가 필드를 보고나서 읽을텐데 보이는 것을 보장하지 않는다는 의미입니다. 결국 원자적 데이터라도 수정된 필드를 보고 읽기 위해서는 동기화의 안정적인 통신이 필요합니다.

❓ 왜 long하고 double은 원자적이지 않은가?

> 같은 primitive 타입이지만 long과 double만 원자직이지 않는 이유는 CPU가 처리하는 기본 단위인 워드보다 길이가 길기 때문입니다.
> 윈도우를 볼 때 32bit, 64bit를 볼 수 있는데 이 때 말하는 bit가 CPU의 기본 단위인 워드를 말합니다.
>
> long 과 double 모두 8바이트로 64비트인데 보통 컴퓨터에서 4바이트를 1워드로 사용했었고 JVM도 보통 4바이트를 1워드로 사용한다고 합니다.
> 즉 JVM = 4바이트이며 long과 double = 8바이트이기 때문에 메모리 할당을 한 번에 해줄 수 없어 원자적이지 못합니다.

동기화 방법 및 예시

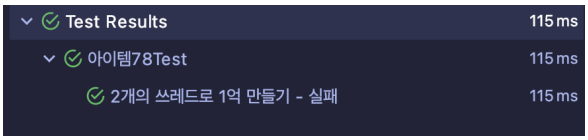
Tread2개로 1억을 만드는 예시

- sum변수가 인스턴스 변수로 공유되고 있다. 이 상태에서 두 개의 쓰레드로 1억을 만들어야한다.
- 하지만 실제로는 2개의 쓰레드가 공유 변수를 건드리기 때문에 1억이 아니라 매번 다른 값이 나오게 된다.
- 이때 같은 공유 메모리에 write하는 행위를 Data Race라고 한다.

해결 방법

1. `synchronized`를 사용한다.
2. Atomic 패키지를 사용해 Lock을 쓰지 않고 sum+= 부분을 Atomic하게 만들어준다.
3. 피터슨 알고리즘을 이용해 직접 Lock 알고리즘을 구현한다.

문제 코드



```java
class ThreadTest {
    private int sum = 0;

    @DisplayName("2개의 쓰레드로 1억 만들기 - 실패")
    @Test
    void 두개쓰레도1억만들기_실패() throws InterruptedException {
        Thread thread1 = new Thread(this::workerThread);
        Thread thread2 = new Thread(this::workerThread);

        thread1.start();
        thread2.start();
        thread1.join();
        thread2.join();

        Assertions.assertThat(sum).isNotEqualTo(100_000_000);
    }

    private void workerThread(){
        for (int i = 0; i < 25_000_000; i++) sum+= 2;
    }
}
```

쓰레드 2개로 1억 만들기 실패 문제 코드

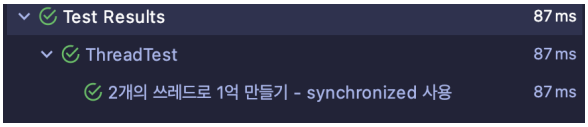
1. **synchronized 해결 방법**

```java
@DisplayName("2개의 쓰레드로 1억 만들기 - synchronized 사용")
@Test
void 두개쓰레도1억만들기_성공1() throws InterruptedException {
    Thread thread1 = new Thread(this::workerThreadSync);
    Thread thread2 = new Thread(this::workerThreadSync);

    thread1.start();
    thread2.start();
    thread1.join();
    thread2.join();

    Assertions.assertThat(sum).isEqualTo(100_000_000);
}

private synchronized void workerThreadSync(){
    for (int i = 0; i < 25_000_000; i++) sum+= 2;
}
```

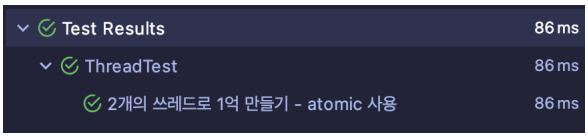


> 공유 변수를 건드리는 쪽에 `synchronized` 키워드를 붙이면 객체가 가진 고유 락으로 동시성 문제를 해결할 수 있다.

2. **volatile 사용**

> `volatile` 키워드를 사용해서 해결할 수 있지만 이 경우에는 하나의 쓰레드만 read하는 상황에서 사용해야 하며 성능도 생각해야 한다.
> 즉 쓰레드가 증가 연산자 처럼 필드를 읽고 수정하는 연산을한다면 두 번째 쓰레드가 비집고 들어와서 새로운 값을 저장해버릴수도 있다.

3. **Atomic 패키지 해결 방법**



```java
private AtomicInteger atomicSum = new AtomicInteger();

@DisplayName("2개의 쓰레드로 1억 만들기 - atomic 사용")
@Test
void 두개쓰레도1억만들기_성공2() throws InterruptedException {
    Thread thread1 = new Thread(this::workerThreadSync);
    Thread thread2 = new Thread(this::workerThreadSync);

    thread1.start();
    thread2.start();
    thread1.join();
    thread2.join();

    Assertions.assertThat(sum).isEqualTo(100_000_000);
}

private void workerThreadAtomic(){
    for (int i = 0; i < 25_000_000; i++) atomicSum.addAndGet( delta: 2);
}
```

> Atomic 패키지를 사용함으로써 정수 값들을 동기화 시킬 수 있습니다.
> volatile이 가지고 있던 단점도 커버하며 성능 또한 좋습니다.
> 그 이유는 메모리에 저장된 값과 CPU Cache에 저장된 값을 비교해 동일한 경우에만 update를 수행하기 때문입니다.

4. **피터슨 알고리즘 사용**

> 2개의 쓰레드일때 boolean값 flag를 직접 만들어서 Lock을 걸어줄 수 있는 피터슨 알고리즘이 있습니다.
> 즉 쓰레드마다 flag를 가지고 계속 while문 돌려서 사용가능한지 판별하는 겁니다.
>
> 그러나 현재의 CPU는 순차적으로 쓰레드를 실행시키지 않아 피터슨 알고리즘이 먹히지 않습니다. 피터슨 알고리즘과 같이 일반적인 프로그래밍 방식으로는 멀티 쓰레드에서 안정적으로 돌아가는 프로그램을 만들 수 없습니다.

아이템 79. 과도한 동기화는 피하라.

❓ 과도한 동기화란?

- 응답 불가와 안전 실패를 피하려면 동기화 메서드나 동기화 블럭 안에서는 제어를 절대로 클라이언트에게 양도하면 안된다.
- 예를 들어, 동기화된 영역 안에서는 재정의할 수 있는 메서드를 호출하면 안되고, 클라이언트가 넘겨준 함수 객체를 호출해서도 안된다.
- 이러한 애들은 외계인이라고 생각하면 된다. 즉 메서드가 무엇을 할지 전혀 모르기 때문이다.
- 따라서 교착상태나 예외, 데이터를 훼손할 수도 있다.

코드 예시

- Observer Paterrn 으로 구현된 Set을 작성
  - 객체의 상태변화를 관찰하는 관찰자들, 즉 옵저버들의 목록을 객체에 등록하여 상태 변화가 있을 때마다 메서드 등을 통해 객체가 직접 목록의 각 옵저버에게 통지하도록 하는 디자인 패턴이다.
  - 주로 분산 이벤트 핸들링 시스템을 구현하는데 사용된다. 발행/구독 모델로 알려져있기도 하다.
- Set에 원소가 추가되면 알림을 받을 수 있는 방법이며, 원소가 제거할 때 알림 기능은 구현돼 있지 않다.
- ForwardingSet을 이용해 예제코드 작성

```
@FunctionalInterface
public interface SetObserver<E>{
    void added(ObservableSet<E> set, E element);
}
```

```
public class ForwardingSet<E> implements Set<E> {
    private final Set<E> s;

    public ForwardingSet(Set<E> s) {
        this.s = s;
    }
    @Override
    public int size() {
        return 0;
    }

    @Override
```

```
    public boolean isEmpty() {
        return false;
    }

    @Override
    public boolean contains(Object o) {
        return false;
    }

    @Override
    public Iterator<E> iterator() {
        return null;
    }

    @Override
    public Object[] toArray() {
        return new Object[0];
    }

    @Override
    public <T> T[] toArray(T[] a) {
        return null;
    }

    @Override
    public boolean add(E e) {
        return false;
    }

    @Override
    public boolean remove(Object o) {
        return false;
    }

    @Override
    public boolean containsAll(Collection<?> c) {
        return false;
    }

    @Override
    public boolean addAll(Collection<? extends E> c) {
        return false;
    }

    @Override
    public boolean retainAll(Collection<?> c) {
        return false;
    }

    @Override
    public boolean removeAll(Collection<?> c) {
        return false;
    }

    @Override
    public void clear() {

    }
}
```

```
public class ObservableSet<E> extends ForwardingSet<E>{
    public ObservableSet(Set<E> set) { super(set); }

    private final List<SetObserver<E>> observers
            = new ArrayList<>();

    public void addObserver(SetObserver<E> observer) {
        synchronized(observers) {
            observers.add(observer);
        }
    }

    public boolean removeObserver(SetObserver<E> observer) {
        synchronized(observers) {
            return observers.remove(observer);
        }
    }

    private void notifyElementAdded(E element) {
        synchronized(observers) {
            for (SetObserver<E> observer : observers)
                observer.added(this, element);
        }
    }

    @Override
    public boolean add(E element) {
        boolean added = super.add(element);
        if (added)
            notifyElementAdded(element);
        return added;
    }

    @Override
```

```
    public boolean addAll(Collection<? extends E> c) {
        boolean result = false;
        for (E element : c)
            result |= add(element);  // notifyElementAdded를 호출한다.
        return result;
    }
}
```

`notifyElementAdded` 메서드를 보면 동기화 영역안에서 콜백 인터페이스의 인스턴스를 사용해 메서드를 호출하는데, 전달하는 함수로 다음과 같은 코드를 보내면 어떻게 될까 한 번 생각해보면 좋다.

```
public static void main(String[] args) {
        ObservableSet<Integer> set =
                new ObservableSet<>(new HashSet<>());

        set.addObserver(new SetObserver<Integer>() {
            @Override
            public void added(ObservableSet<Integer> set, Integer element) {
                System.out.println(element);
                if  (element == 23) {
                    set.removeObserver(this);
                }
            }
        });

        for (int i = 0; i < 100; i++)
            set.add(i);
    }
```

e가 23일때 구독을 해제하고 종료할 것 같지만 실제로는 `ConcurrentModificationException` 예외를 던진다.

리스트에서 원소를 제거하려 하는데, 동시에 리스를 순회하려 하기 때문이다. 즉 `notifyElementAdded` 메서드에서 순회로직은 동기화 블록 안에 있지만, 정작 콜백 메서드로 수정을 하는 것을 막지 못하기 되는 것이다.

교착상태와 재진입

```
public static void main(String[] args) {
    ObservableSet<Integer> set =
            new ObservableSet<>(new HashSet<>());

    set.addObserver(new SetObserver<Integer>() {
        @Override
        public void added(ObservableSet<Integer> set, Integer element) {
            System.out.println(element);
            if  (element == 23) {
                ExecutorService executorService = Executors.newSingleThreadExecutor();
                try{
                    executorService.submit(() -> set.removeObserver(this)).get();
                }catch (ExecutionException | InterruptedException e){
                    throw new AssertionError(e);
                }finally {
                    executorService.shutdown();
                }
            }
        }
    });
}
```

1. set.removeObserver 메서드를 호출하면 관찰자를 잠그려 한다.
2. 하지만, 락을 얻을 수 없다. 메인 스레드에서 락을 쥐고 있기 때문이다.
3. 이 때 동시에 메인 스레드에서는 백그라운드 스레드가 관찰자를 제거하기를 기다리고 있다.
4. 교착상태 발생

해결책

1. 콜백 메서드를 동기화 바깥으로 옮기자.

   즉 문제가 될 여지가 있는 코드를 동기화 블록 바깥으로 옮기는 것이다.

   ```
   private void notifyElementAdded(E element) {
       List<SetObserver<E>> snapshot = null;
       synchronized (observers) {
         snapshot = new ArrayList<>(observers);
       }
       for (SetObserver<E> observer : observers) {
         observer.added(this, element);
       }
   }
   ```

2. CopyOnWriteArrayList

> java.util.concurrent 에서 제공하는 CopyOnWriteArrayList 클래스는 이런 문제를 해결하기 위해 설계된 객체로, ArrayList를 구현하되 내부를 변경하는 작업은 늘 복사본을 만들어 수행하도록 구현되었다. 다른 용도로는 끔찍하게 느리지만, 순회가 대부분이고 간혹 수정할 일이 있는 관찰자 리스트 용도로는 적절하다

가변 클래스 작성 팁

1. 동기화를 하지 말고, 클래스를 동시에 사용해야 하는 클래스가 외부에서 동기화 하도록 하자.
   - java.util
2. 동기화를 내부에서 수행해 쓰레드 안전한 클래스로 만들자.
   - 락 분할
   - 락 스트라이핑
   - 비차단 동시성 제어

아이템 80. 스레드보다는 실행자, 태스크, 스트림을 애용하라.

실행자 프레임워크

> `java.util.concurrent` 패키지에는 인터페이스 기반의 유연한 태스크 실행 기능을 담은 실행자 프레임워크(Executor Framework)가 있다. 과거에는 단순한 작업 큐(work queue)를 만들기 위해서 수많은 코드를 작성해야 했는데, 이제는 아래와 같이 간단하게 작업 큐를 생성할 수 있다.

```
// 큐를 생성한다.
ExecutorService exec = Executors.newSingleThreadExecutor();

// 태스크 실행
exec.execute(runnable);

// 실행자 종료
exec.shutdown();
```

- 특정 태스크가 완료되기를 기다릴 수 있다. `submit().get()`

```
ExecutorService exec = Executors.newSingleThreadExecutor();
exec.submit(()  -> s.removeObserver(this)).get(); // 끝날 때까지 기다린다.
```

- 태스크 모음 중에서 어느 하나(`invokeAny`) 혹은 모든 태스크(`invokeAll`)가 완료되는 것을 기다릴 수 있다.

```
List<Future<String>> futures = exec.invokeAll(tasks);
System.out.println("All Tasks done");

exec.invokeAny(tasks);
System.out.println("Any Task done");
```

- 실행자 서비스가 종료하기를 기다린다. `awaitTermination`

```
Future<String> future = exec.submit(task);
exec.awaitTermination(10, TimeUnit.SECONDS);
```

- 완료된 태스크들의 결과를 차례로 받는다. `ExecutorCompletionService`

```
final int MAX_SIZE = 3;
ExecutorService executorService = Executors.newFixedThreadPool(MAX_SIZE);
ExecutorCompletionService<String> executorCompletionService = new ExecutorCompletionService<>(executorService);

List<Future<String>> futures = new ArrayList<>();
futures.add(executorCompletionService.submit(() -> "madplay"));
futures.add(executorCompletionService.submit(() -> "kimtaeng"));
futures.add(executorCompletionService.submit(() -> "hello"));

for (int loopCount = 0; loopCount < MAX_SIZE; loopCount++) {
    try {
        String result = executorCompletionService.take().get();
        System.out.println(result);
    } catch (InterruptedException e) {
        //
    } catch (ExecutionException e) {
        //
    }
```

```
}
executorService.shutdown();
```

- 태스크를 특정 시간에 혹은 주기적으로 실행하게 한다. `ScheduledThreadPoolExecutor`

```
ScheduledThreadPoolExecutor executor = new ScheduledThreadPoolExecutor(1);

executor.scheduleAtFixedRate(() -> {
    System.out.println(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss")
            .format(LocalDateTime.now()));
}, 0, 2, TimeUnit.SECONDS);

// 2019-09-30 23:11:22
// 2019-09-30 23:11:24
// 2019-09-30 23:11:26
// 2019-09-30 23:11:28
// ...
```

ThreadPool 종류

> `Executors.newCachedThreadPool`은 가벼운 프로그램을 실행하는 서버에 적합하다. 요청받은 태스크를 큐에 쌓지 않고 바로 처리하며 사용 가능한 스레드가 없다면 즉시 스레드를 새로 생성하고 처리한다. 서버가 무겁다면 CPU 이용률이 100%로 치닫고 새로운 태스크가 도착할 때마다 다른 스레드를 생성하며 상황이 더 악화될 것이다.

🌟 따라서 무거운 프로덕션 서버에는 `Executors.newFixedThreadPool`을 선택하여 스레드 개수를 고정하는 것이 좋다.

스레드를 직접 다루지 말자

> 작업 큐를 직접 만들거나 스레드를 직접 다루는 것도 일반적으로 삼가야 한다. 스레드를 직접 다루지말고 실행자 프레임워크를 이용하자. 그러면 작업 단위와 실행 매커니즘을 분리할 수 있는다. 작업 단위는 `Runnable`과 `Callable`로 나눌 수 있다. Callable은 Runnable과 비슷하지만 값을 반환하고 임의의 예외를 던질 수 있다.

자바 7부터 실행자 프레임워크는 포크-조인(fork-join) 태스크를 지원하도록 확장됐다.

ForkJoinTask의 인스턴스는 작은 하위 태스크로 나뉠 수 있고 ForkJoinPool을 구성하는 스레드들이 이 태스크들을 처리하며, 일을 먼저 끝낸 스레드가 다른 스레드의 남은 태스크를 가져와 대신 처리할 수도 있다.

이렇게 하여 최대한의 CPU 활용을 뽑아내어 높은 처리량과 낮은 지연시간을 달성한다. 병렬 스트림도 이러한 ForkJoinPool을 이용하여 구현되어 있다.

아이템 81. wait와 notify 보다는 동시성 유틸리티를 애용하라.

동시성 컬렉션

> 동시성 컬렉션은 List, Queue, Map과 같은 표준 컬렉션 인터페이스에 동시성을 가미한 고성능 컬렉션이다.

- 내부적으로 동기화를 각자의 내부에서 수행한다.
- 동시성 컬렉션에서 동시성을 무력화하는건 불가능하다.
- 외부에서 락을 추가로 사용하면 오히려 속도가 느려진다.
  - 동기화를 중첩으로 사용한다고 더 안전해지고 그런건 아니기에 불필요하다.
- 동시성을 무력화하지 못하기에 여러 메서드가 원자적으로 묶여 호출하지는 못하고, 여러 기본 동작을 하나의 원자적 동작으로 묶는 상태 의존적 수정 메서드들이 추가 됐다. (Map의 putIf(key ,value))

동기화 컬렉션보다 안전과 속도 모두 개선됐기에 동시성 컬렉션을 사용하는걸 권장한다.

동기화 장치는 쓰레드가 다른 쓰레드를 기다릴 수 있게 한다.

> 그래서 작업들을 조율할 수 있게 해주는데, 예를들어 Queue를 확장한 BlockingQueue도 확장된 기능중 take라는 기능을 볼 수 있는데, 큐의 첫 번째 원소를 꺼내는 기능으로 만약 큐가 비었다면 새로운 원소가 추가될 때까지 기다린다. 그렇기에 BlockingQueue는 작업 큐로 쓰기 적당하고 ThreadPoolExecutor나 실행자 서비스 구현체에서 BlockingQueue를 사용한다.

그 밖에 또 자주 사용되는 동기화 장치로 다음과 같은 장치들이 있다.

- CountDownLatch

- Semaphore
- CyclicBarrier
- Exchanger
- Phaser

```
public static long time(Executor executor, int concurrency, Runnable action)
            throws InterruptedException {

    CountDownLatch ready = new CountDownLatch(concurrency);
    CountDownLatch start = new CountDownLatch(1);
    CountDownLatch done = new CountDownLatch(concurrency);

    for (int i = 0; i < concurrency; i++) {
        executor.execute(() -> {
            ready.countDown();
            try {
                start.await();
                action.run();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            } finally {
                done.countDown();
            }
        });
    }

    ready.await();
    long startNanos = System.nanoTime();
    start.countDown();
    done.await();
    return System.nanoTime() - startNanos;
}
```

- 스레드 기아 교착상태 코드
- 위와 같은 메서드에 넘겨진 실행자는 concurrency 매개변수로 동시성 수준만큼의 쓰레드를 생성할 수 있어야하는데, 쓰레드 생성을 하지 못하면 이 메서드는 끝나지 않고 교착상태에 걸리게 되는데, 이런 상태를 쓰레드 기아 교착 상태라고 한다.

아이템 82. 스레드 안전성 수준을 문서화하라.

> API 문서에 스레드 안정성에 대한 내용이 없으면 사용자는 동기화를 부족하게 하거나 과하게 하는 경우가 생길 수 있다. 멀티스레드 환경에서도 API를 안전하게 사용하게 하려면 클래스가 지원하는 스레드 안전성 수준을 명시해야 한다.

쓰레드 안전성 수준이 높은 순

1. **불변 (immutable)**
   - 이 클래스의 인스턴스는 마치 상수와 같아서 외부 동기화가 필요없다.
   - 예) String, Long, BigInteger 등..
2. **무조건적 스레드 안전 (unconditionally thread-safe)**
   - 이 클래스의 인스턴스는 수정될 수 있으나, 내부에서 충실히 동기화하여 별도의 외부 동기화가 필요없다.
   - synchronized 메서드가 아닌 비공개 락 객체를 사용한다.
   - 예) AtomicLong, ConcurrentHashMap
3. **조건부 스레드 안전 (conditionally thread-safe)**
   - 무조건적 스레드 안전과 같으나 일부 메서드는 동시에 사용하려면 외부 동기화가 필요하다
   - 그래서 주의해서 문서화해야 한다.
     - 어떤 순서로 호출할 때 외부 동기화가 필요한지, 그 순서로 호출하려면 어떤 락을 얻어야 하는지 등
     - 클래스의 스레드 안정성은 보통 클래스의 문서화 주석에 기재하지만, 독특한 메서드라면 해당 메서드에 문서화하자.
   - 예) Collections.synchronizedMap, Collections.synchronizedSet 등등이 반환한 컬렉션들
4. **스레드 안전하지 않음 (not thread-safe)**
   - 클래스의 인스턴스가 수정될 수 있다
   - 동시에 사용하려면 클라이언트가 별도의 동기화를 수행해야 한다.
   - 예) ArrayList, HashMap 등의 기본 컬렉션
5. **스레드 적대적 (thread-hostile)**
   - 이 클래스는 모든 메서드 호출을 외부 동기화로 감싸더라도 멀티스레드 환경에서 안전하지 않다.

- 고의로 만드는 사람은 없지만 동시성을 고려하지 않고 작성하다보면 우연히 만들어질 수 있다.
- 예) generateSerialNumber에서 내부 동기화를 생략했을 때

❓ 외부에서 사용할 수 있는 락을 제공하면?

```
/**
 *  synchronizedMap이 반환한 맵의 컬렉션 뷰를 순회하려면 반드시 그 맵의 락으로 사용해 수동으로 동기화하라.
 * <pre>
 *  Map m = Collections.synchronizedMap(new HashMap());
 *      ...
 *  Set s = m.keySet();  // 동기화 블록 밖에 있어도 된다.
 *      ...
 *  synchronized (m) {  // s가 아닌 m을 사용해 동기화해야 한다.
 *      Iterator i = s.iterator(); // 동기화 블록 안에 있어야 한다.
 *      while (i.hasNext())
 *          foo(i.next());
 *  }
 * </pre>
 * 이대로 하지 않으면 동작을 예측할 수 없다.
 */
public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m) {
    return new SynchronizedMap<>(m);
}
```

- 장점 : 클라이언트에서 일련의 메서드를 원자적으로 수행할 수 있다.
- 단점
  - 내부에서 처리하는 고성능 동시성 제어 메커니즘과 혼용할 수 없다 -> **동시성 컬렉션과는 함께 사용할 수 없다**
  - 클라이언트가 DOS공격(공개된 락을 오래 쥐고 놓지 않음)을 수행할 수도 있다.
  - **DOS공격을 막으려면 비공개 락 객체를 사용해야 한다.**

❓ 비공개 락 객체가 뭔데?

```
//락 객체가 교체되는 걸 방지하기 위해 final, 외부에서 못보게 private
private final Object lock = new Object();

public void foo() {
  synchronized(lock) {
    ...
  }
}
```

- 클라이언트가 객체의 동기화에 관여할 수 없다
- 무조건적 스레드 안전 클래스에서만 사용할 수 있다. 조건부 스레드 안전 클래스에서는 사용할 수 없다
- 상속용으로 설계한 클래스에 특히 잘 맞는다.
  - 하위 클래스에서 기반 클래스의 락을 사용한다면, 서로가 서로를 훼방놓는 상태에 빠진다. 기반 클래스의 락에 애초에 접근할 수 없으므로 상속용으로 좋다.

아이템 83. 지연 초기화는 신중히 사용하라.

지연 초기화

- `클래스` 혹은 `인스턴스 생성` 시의 **초기화 비용**은 줄지만, 지연 초기화하는 **필드에 접근하는 비용**은 커진다.
- `지연 초기화하려는 필드`들 중 결국 **초기화가 이뤄지는 비율**에 따라, **실제 초기화에 드는 비용**에 따라, **초기화된 각 필드를 얼마나 빈번히 호출**하느냐에 따라 `지연 초기화`가 실제로는 **성능을 느려지게 할 수도 있다.**

지연 초기화가 필요한 경우

- 해당 클래스의 인스턴스 중 **그 필드를 사용하는 인스턴스의 비율이 낮은 반면**, 그 **필드를 초기화하는 비용이 크다면** `지연 초기화`가 제 역할을 할 것이다.
- 하지만 위 효과를 확인하기 위해서는 `지연 초기화 작용 전후의 성능을 측정`해보는 것이다.

멀티 스레드 환경의 지연 초기화

- 멀티스레드 환경에서는 지연 초기화를 하기 힘들다.
- 지연 초기화하는 필드를 둘 이상의 스레드가 공유한다면 반드시 동기화를 해야 한다. 아이템 78

Thread Safe 한 지연 초기화

- 대부분의 상황에서 일반적인 초기화가 지연 초기화보다 낫다.

관용구 1: 인스턴스 필드를 선언할 때의 일반적인 초기화

```
class Example {
    // final 한정자를 통한 인스턴스 필드 생성
    private final FieldType field = computeFieldValue();
}
```

관용구 2: 인스턴스 필드의 지연 초기화 (synchronized 접근자를 통한 방식)

- `지연 초기화`가 **초기화 순환성(initialization circularity)** 을 깨뜨릴 것 같으면 `synchronized`를 단 접근자를 사용한다.

```
class Example {
    private final FieldType field;

    private synchronized FieldType getField() {
        if (field == null) {
            field = computeFieldValue();
        }
        return field;
    }
}
```

- 두 관용구(`보통의 초기화`와 `synchronized 접근자를 사용한 지연 초기화`)는 **정적 필드**에도 똑같이 적용된다.
- 이때 `필드`와 `접근자 메서드 선언`에 **static 한정자**를 추가해야 한다.

정적 필드용 지연 초기화 홀더 클래스 관용구

- 성능면에서 `정적 필드의 지연 초기화`가 필요한 경우 **지연 초기화 홀더 클래스 관용구(lazy initialization holder class)** 를 사용한다.
- 클래스는 클래스가 처음 쓰일 때 비로소 초기화된다는 특성을 이용한 관용구

```
class Example {
    private static class FieldHolder {
        static final FieldType field = computeFieldValue();
    }

    private static FieldType getField() { return FieldHolder.field; }
}
```

- getField가 처음 호출되는 순간 FieldHolder.field가 처음 읽히면서, 비로소 FieldHolder 클래스 초기화를 촉발한다.
- getField `메서드가 필드에 접근`하면서 **동기화를 전혀 하지 않으니** 성능이 느려질 이유가 전혀 없다.

❓ 성능이 느려질 이유가 없다?

- 일반적인 VM은 오직 클래스를 초기화할 때만 필드 접근을 동기화할 것이다.
- 클래스 초기화가 끝난 후에는 VM이 동기화 코드를 제거하여, 그 다음부터는 아무런 검사나 동기화 없이 필드에 접근하게 된다.

성능적인 측면에서의 인스턴스 필드 지연 초기화를 위한 이중검사관용구(double-check)

- 이 방법은 초기화된 필드에 접근할 때의 동기화 비용을 없애준다.
- **동작 방식**
  - 필드의 값을 두 번 검사하는 방식
  - **한 번**은 동기화 없이 검사
  - **두 번째**는 동기화하여 검사
  - 두 번째 검사에서도 `필드가 초기화되지 않았을 때`만 **필드를 초기화** 한다.
- **주의 사항**
  - 필드가 초기화된 후로는 동기화하지 않으므로 해당 필드는 반드시 volatile 로 선언해야 한다.

```
class Example {
    private volatile FieldType field;

    private FieldType getField() {
        FieldType result = field; // 초기화 시 한 번만 읽도록 하기 위함
      if(result != null) {
          return result;
        }

      synchronized (this) {
          if(field == null) { // 두 번째 검사 (락 사용)
              field = computeFieldValue();
            }
          return field;
        }
    }
}
```

- **코드 분석**
  - result라는 지역변수의 용도는 필드가 이미 초기화된 상황(일반적인 상황에서)에서는 그 필드를 딱 한번만 읽도록 보장하는 역할을 한다.
  - 반드시 필요하지는 않지만 성능을 높여주고, 저수준 동시성 프로그래밍에 표준적으로 적용되는 더 우아한 방법이다.
- **정적 필드**에 대한 **이중검사 관용구** 적용 가능성
  - **정적 필드**를 `지연 초기화`하기 위해서는 **이중검사**보다 **지연 초기화 홀더 클래스** 방식이 더 낫다.

이중 검사의 특이 유형 2가지

- 상황
  - `반복해서 초기화해도 상관없는` **인스턴스 필드를 지연초기화**해야 하는 경우, `이중검사`에서 **두 번째 검사를 생략**할 수 있다.

단일 검사(single-check) 관용구

- 필드는 volatile로 선언
- 초기화가 중복해서 일어날 수 있다.

```
class Example {
    private volatile FieldType field;
    private FieldType getField() {
        FieldType result = field;
        if(result == null) {
            field = result = computeFieldValue();
        }
        return result;
    }
}
```

짜릿한 단일검사(racy single-check) 도구

- **사용 조건**
  - 모든 스레드가 필드의 값을 다시 계산해도 되고, 필드의 타입이 long과 double을 제외한 다른 기본 타입이라면, 단일검사의 필드 선언에서 volatile 한정자를 없애도 된다.
- **효과**
  - 어떤 환경에서는 필드 접근 속도를 높여주지만, 초기화가 스레드당 최대 한 번 더 이루어질 수 있다.

아주 이례적인 기법으로, 보통은 쓰이지 않는다.

정리

- 여기서 다룬 모든 초기화 기법은 **기본 타입 필드**와 **객체 참조 필드**에 모두 적용할 수 있다.
- **이중검사**와 **단일검사 관용구**를 **수치 기본 타입 필드**에 적용한다면 필드의 값을 **null 대신**(숫자 기본 타입 변수의 기본값인)**0**과 비교하면 된다.
- `대부분`의 필드는 **지연시키지말고 곧바로 초기화**해야 한다.
- **성능** 때문에 혹은 **위험한 초기화 순환**을 막기 위해 꼭 `지연 초기화`를 써야 하는 경우 **올바른 지연 초기화 기법**을 사용한다.
- **인스턴스 필드**에는 **이중검사 관용구**, **정적 필드**에는 **지연 초기화 홀더 클래스 관용구**를 사용한다.
- 반복해 초기화해도 괜찮은 인스턴스 필드에는 **단일검사 관용구**도 고려대상이다.

아이템 84. 프로그램의 동작을 스레드 스케줄러에 기대지 말라.

- 여러 스레드가 실행 중이면 운영체제의 스레드 스케줄러가 어떤 스레드를 얼마나 오래 실행할지 정한다.
- 정상적인 운영체제라면 이 작업을 공정하게 수행하지만 구체적인 스케줄링 정책은 운영체제마다 다를 수 있다.
- **정확성이나 성능이 스레드 스케줄러에 따라 달라지는 프로그램이라면 다른 플랫폼에 이식하기 어렵다.**
- 견고하고 빠릿하고 이식성 좋은 프로그램을 작성하는 가장 좋은 방법은 **실행 가능한 스레드의 평균적인 수를 프로세서 수보다 지나치게 많아지지 않도록 하는 것이다.**
- 실행 가능한 스레드 수를 적게 유지하는 주요 기법은 각 스레드가 무언가 유용한 작업을 완료한 후에는 다음 일거리가 생길 때까지 대기하도록 하는 것이다.
- **스레드는 당장 처리해야 할 작업이 없다면 실행돼서는 안 된다.**
- 스레드는 절대 바쁜 대기(busy waiting) 상태가 되면 안 된다.
- 공유 객체의 상태가 바뀔 때까지 쉬지않고 검사해서는 안 된다는 뜻이다.

❓ 바쁜 대기란?
참고

🌟 어떠한 특정 공유자원에 대하여 두 개 이상의 프로세스나 스레드가
그 이용 권한을 획득하고자 하는 동기화 상황에서 그 권한 획득을 위한 과정에서 일어나는 현상이다.
대부분의 경우에 스핀락(Spin-lock)과 이것을 동일하게 생각하지만, 엄밀히 말하자면 스핀락이 바쁜 대기 개념을 이용한 것이다.

즉 원하는 자원을 얻기 위해 기다리는 것이 아니라 권한을 얻을 때까지 확인하는 것
→ 권한 획득을 위해 많은 CPU를 낭비한다는 단점이 존재함

Thread.yield

- 다른 쓰레드에게 작업(실행)을 양보하고 실행 대기 상태가 된다.

```
class MyThread extends Thread {

    @Override
    public void run() {
        for (int i = 0; i < 5; i++)
            System.out.println(Thread.currentThread().getName() + " in control");
    }
}

public class Main {
    public static void main(String[] args) {
        MyThread t = new MyThread();
        t.start();

        for (int i = 0; i < 5; i++) {
            // Control passes to child thread
            Thread.yield();

            // After execution of child Thread
            // main thread takes over
            System.out.println(Thread.currentThread().getName() + " in control");
        }
    }
}

main in control
Thread-0 in control
Thread-0 in control
Thread-0 in control
Thread-0 in control
Thread-0 in control
main in control
main in control
main in control
main in control


main in control
main in control
Thread-0 in control
Thread-0 in control
Thread-0 in control
main in control
Thread-0 in control
Thread-0 in control
main in control
main in control
```

- 특정 스레드가 다른 스레드들과 비교해 CPU 시간을 충분히 얻지 못해서 간신히 돌아가는 프로그램을 보더라도 **Thread.yield를 써서 문제를 고쳐보려는 유혹을 떨쳐내자.**
- 증상이 어느정도 호전될 수 있으나 이식성은 그렇지 않을 것이다.
- **Thread.yield는 테스트할 수단도 없다.**
- 차라리 애플리케이션 구조를 바꿔 동시에 실행 가능한 스레드 수가 적어지도록 조치해주자.

정리

- 프로그램의 동작을 스레드 스케줄러에 기대지 말자.
- 견고성과 이식성을 모두 해치는 행위다.
- 같은 이유로, Thread.yield와 스레드 우선순위에 의존해서도 안 된다.
- 이 기능들은 스레드 스케줄러에 제공하는 힌트일 뿐이다.
- 스레드 우선순위는 이미 잘 동작하는 프로그램의 서비스 품질을 높이기 위해 드물게 쓰일 수는 있지만,
간신히 동작하는 프로그램을 '고치는 용도'로 사용해서는 절대 안 된다.

🚪 10장 예외

🚪 12장 직렬화