동시성

동시성 문제

멀티스레드 환경에서 여러 개의 스레드가 동시에 실행되는 상황에서 발생하는 문제들을 의미한다.
운영체제의 스케줄러가 각 스레드를 번갈아가며 실행한다.

1. **경쟁 조건**
    - 두 개 이상의 스레드가 동시에 공유 자원에 접근하여 발생하는 상황
    - 여러 스레드가 동시에 같은 변수를 수정하거나, 같은 파일을 동시에 쓰기를 시도하는 경우
2. **교착상태 - 데드락**
    - 두 개 이상의 스레드가 서로가 가지고 있는 자원을 기다리며 무한히 대기하는 상황
    - 스레드 A가 자원 1을 가지고 있고, 스레드 B가 자원 2를 가지고 있고, 스레드 A가 자원 2를 요청하고, 스레드 B가 자원 1을 요청하는 경우
3. **동기화 문제**
    - 여러 개의 스레드가 공유 자원에 접근할 때, 동기화가 제대로 이루어지지 않아 발생하는 상황
    - 스레드가 공유 자원을 읽고 쓰는 중간에 다른 스레드가 동일한 자원에 접근하는 경우
4. **메모리 일관성**
    - 멀티스레드 환경에서 메모리의 일관성을 유지하기 위해 필요한 메모리 접근 규칙을 제대로 지키지 않아 발생하는 상황

동시성 제어 방법

락

- 동시성 문제를 해결하기 위한 동기화 기술
- 여러 개의 스레드가 공유된 자원에 접근할 때 상호간에 동시에 접근하지 못하도록 제어

- 락은 한 스레드가 특정 자원에 락을 획득하면, 다른 스레드들은 해당 자원에 대한 접근을 차단
- 이를 통해 스레드 간의 충돌이나 경쟁 상태를 방지하여 동시성 문제를 해결할 수 있습니다.

비관적 락

- 트랜잭션이 시작될 때부터 해당 데이터에 대한 잠금을 설정하고, 트랜잭션이 끝날 때까지 유지하는 방식
- 데이터 일관성을 보장할 수 있지만, 동시성 처리 성능이 저하될 수 있습니다.

낙관적 락

- 데이터의 일관성을 보장하지 않고, 데이터를 읽기만 할 때는 잠금을 걸지 않고, 데이터를 업데이트할 때만 잠금을 걸어 데이터 일관성을 유지하는 방식
- 데이터의 일관성을 유지하면서도 동시성 처리 성능을 개선할 수 있다.
- 여러 개의 프로세스나 스레드가 동시에 데이터를 업데이트하는 경우 충돌이 발생할 수 있다.
- 충돌을 처리하는 방식에 따라 성능이 저하될 수 있다.

하지만 락이 남발되면 성능 저하나 데드락 같은 문제가 발생할 수 있다.

- 락을 사용할 때는 동기화의 범위를 최소화하여 경합 상태를 줄이고, 락을 확보하는 시간을 최소화하여 성능을 향상시키는 것이 중요하다.
- 락을 사용할 때에는 교착상태와 같은 부작용에 주의하여 안전하고 효율적인 동시성 제어를 구현해야 한다.
- 가능하다면 락 없이 동시성 문제를 해결할 수 있는 다른 방법들을 고려하는 것이 좋다.
- 비동기 프로그래밍, 락을 대체할 수 있는 동시성 컬렉션이 있다.

***경합 상태**

- 두 스레드가 공유 변수에 동시에 접근할 때 발생

동시성 문제 발생 코드

```
private int saveNumber = 0;

@Test
public void concurrency() throws InterruptedException {
    log.info("concurrency start");

    Runnable test1 = () -> {
        start(1);
    };

    Runnable test2 = () -> {
        start(2);
    };

    Thread threadA = new Thread(test1);
    threadA.setName("thread-A");

    Thread threadB = new Thread(test2);
    threadB.setName("thread-B");

    threadA.start();
    sleep(100);
    threadB.start();
    sleep(3000);

    log.info("concurrency end");
}

private int start(int number) {
  log.info("start : {}", number);
  log.info("현재 number = {} , saveNumber = {} ", number, save
  saveNumber = number;
  try {
      sleep(1000);
  } catch (InterruptedException e) {
      throw new RuntimeException(e);
```

```
    }
    log.info("saveNumber : {}", saveNumber);

    return saveNumber;
}
```

sleep 0.1초 설정

```
12:46:56.207 [Test worker] INFO com.okestro.vista.portal.service.ServerServiceTest -- concurrency start
12:46:56.212 [thread-A] INFO com.okestro.vista.portal.service.ServerServiceTest -- start : 1
12:46:56.214 [thread-A] INFO com.okestro.vista.portal.service.ServerServiceTest -- 현재 number = 1 , saveNumber = 0
12:46:56.317 [thread-B] INFO com.okestro.vista.portal.service.ServerServiceTest -- start : 2
12:46:56.317 [thread-B] INFO com.okestro.vista.portal.service.ServerServiceTest -- 현재 number = 2 , saveNumber = 1
12:46:57.216 [thread-A] INFO com.okestro.vista.portal.service.ServerServiceTest -- saveNumber : 2
12:46:57.323 [thread-B] INFO com.okestro.vista.portal.service.ServerServiceTest -- saveNumber : 2
12:46:59.320 [Test worker] INFO com.okestro.vista.portal.service.ServerServiceTest -- concurrency end
```

sleep를 3초로 설정

```
12:45:59.101 [Test worker] INFO com.okestro.vista.portal.service.ServerServiceTest -- concurrency start
12:45:59.104 [thread-A] INFO com.okestro.vista.portal.service.ServerServiceTest -- start : 1
12:45:59.106 [thread-A] INFO com.okestro.vista.portal.service.ServerServiceTest -- 현재 number = 1 , saveNumber = 0
12:46:00.111 [thread-A] INFO com.okestro.vista.portal.service.ServerServiceTest -- saveNumber : 1
12:46:02.110 [thread-B] INFO com.okestro.vista.portal.service.ServerServiceTest -- start : 2
12:46:02.110 [thread-B] INFO com.okestro.vista.portal.service.ServerServiceTest -- 현재 number = 2 , saveNumber = 1
12:46:03.116 [thread-B] INFO com.okestro.vista.portal.service.ServerServiceTest -- saveNumber : 2
12:46:05.113 [Test worker] INFO com.okestro.vista.portal.service.ServerServiceTest -- concurrency end
```

문제

thread A의 saveNumber는 1을 조회해야 하지만 현재 2를 조회함으로서 동시성 문제가 발생하고 있다.

해결방법

이와 같은 동시성 문제를 해결 할 수 있는 방법으론 synchronized 키워드를 이용하여 해결 할 수 있다.

synchronized - 비관적 락

- synchronized 메서드를 호출한다면 이 메서드가 종료될 때까지 다른 쓰레드가 이 메서드를 수행 할 수 없다.

```
private int saveNumber = 0;

 @Test
 public void concurrency() throws InterruptedException {
     log.info("concurrency start");

     Runnable test1 = () -> {
         start(1);
     };

     Runnable test2 = () -> {
         start(2);
     };

     Thread threadA = new Thread(test1);
     threadA.setName("thread-A");

     Thread threadB = new Thread(test2);
     threadB.setName("thread-B");

     threadA.start();
     sleep(100);
     threadB.start();
     sleep(3000);

     log.info("concurrency end");
 }

 private synchronized int start(int number) {
     log.info("start : {}", number);
     log.info("현재 number = {} , saveNumber = {} ", number,
     saveNumber = number;
     try {
         sleep(1000);
     } catch (InterruptedException e) {
         throw new RuntimeException(e);
     }
     log.info("saveNumber : {}", saveNumber);
```

```
        return saveNumber;
    }
```

```
05:09:41.753 [Test worker] INFO com.study.spa.concurrency.ConcurrencyTest - concurrency start
05:09:41.757 [thread-A] INFO com.study.spa.concurrency.ConcurrencyServiceTest - number start
05:09:41.758 [thread-A] INFO com.study.spa.concurrency.ConcurrencyServiceTest - 현재 number = 1 , 저장 saveNumber = 0
05:09:42.787 [thread-A] INFO com.study.spa.concurrency.ConcurrencyServiceTest - 조회 saveNumber = 1
05:09:42.792 [thread-B] INFO com.study.spa.concurrency.ConcurrencyServiceTest - number start
05:09:42.792 [thread-B] INFO com.study.spa.concurrency.ConcurrencyServiceTest - 현재 number = 2 , 저장 saveNumber = 1
05:09:43.798 [thread-B] INFO com.study.spa.concurrency.ConcurrencyServiceTest - 조회 saveNumber = 2
05:09:44.867 [Test worker] INFO com.study.spa.concurrency.ConcurrencyTest - concurrency end
```

thread A의 saveNumber는 1을 조회하고 thread B의 saveNumber는 2를조회한다.

synchronized를 이용해서 lock을 걸어서 동기화를 처리할 수 있었고 결과도 정상적으로 볼 수 있었다.

자바에서 동시성 문제 처리를 위해서 synchronized 키워드를 사용하여 하나의 쓰레드가 수행 중인 메서드의 작업을 끝내기 전까지 다른 쓰레드들은 동일한 작업을 수행하지 못하도록 만들었다.

DB 단에서 락을 거는 방법

먼저 동시성 문제 발생 코드이다.

```
@Test
public void concurrency() throws InterruptedException {
    log.info("concurrency start");

    Runnable test1 = () -> {
              concurrencyRepository.jpaconcurrency(1L);
      };

    Runnable test2 = () -> {
        concurrencyRepository.jpaconcurrency(1L);
    };
```

```
    Thread threadA = new Thread(test1);
    threadA.setName("thread-A");

    Thread threadB = new Thread(test2);
    threadB.setName("thread-B");

    threadA.start();
    sleep(200);
    threadB.start();

    log.info("concurrency end");
}
```

```
@Transactional
public Concurrency jpaconcurrency(Long id) {
   log.info("jpaConcurrency start");
   Concurrency concurrency = concurrencyRepository.findById(i
   log.info("현재 count = {}" , concurrency.getCount());
   // count ++;
   concurrecny.setCount();
   log.info("변경 count = {}" , concurrency.getCount());

   return concurrency;
}
```

```
[      thread-A] c.study.spa.service.ConcurrencyService   : 현재 count = 0
[      thread-B] c.study.spa.service.ConcurrencyService   : 현재 count = 0
[      thread-A] c.study.spa.service.ConcurrencyService   : 변경 count = 1
[      thread-B] c.study.spa.service.ConcurrencyService   : 변경 count = 1
```

동시성 문제가 발생해서 thread-B의 변경 값이 thread-A의 값과 같은 걸 볼 수 있다.

낙관적 락

- 트랜잭션은 충돌하지 않을거라고 가정하고 동시성 문제가 발생 했을때 처리하는 방법

```java
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;

private int count;

@Version
private Integer version;
```

엔티티에 @Version을 컬럼을 추가하여 버전 관리를 진행한다.

```
select
    concurrenc0_.id as id1_0_0_,
    concurrenc0_.count as count2_0_0_,
    concurrenc0_.version as version3_0_0_
from
    concurrency concurrenc0_
where
    concurrenc0_.id=?
```

```
update
    concurrency
set
    count=?,
    version=?
where
    id=?
    and version=?
```

비관적 락

- 트랜잭션은 충돌이 발생한다고 가정하고 락을 거는 방법
- DB가 제공하는 락을 거는 방법(select for update)

JPA에서 비관적 락을 걸기 위한 3가지 방법이 있다.

1. LockModeType.PESSIMISTIC_WRITE - 배타적 잠금

- 한 트랜잭션이 작업을 수행 중 일때 트랜잭션이 완료될때까지 다른 트랜잭션은 작업을 진행 할 수 없다.

```java
@Lock(LockModeType.PESSIMISTIC_WRITE)
@QueryHints({@QueryHint(name = "javax.persistence.lock.timeout", value = "1000")})
Optional<Concurrency> findById(Long id);
```

2.LockModeType.PESSIMISTIC_FORCE_INCREMENT

- Version 정보를 사용하며 비관적 락을 건다.

```
@Lock(LockModeType.PESSIMISTIC_FORCE_INCREMENT)
@QueryHints({@QueryHint(name = "javax.persistence.lock.timeout", value = "1000")})
Optional<Concurrency> findById(Long id);
```

```
    select
        concurrenc0_.id as id1_0_0_,
        concurrenc0_.count as count2_0_0_,
        concurrenc0_.version as version3_0_0_
    from
        concurrency concurrenc0_
    where
        concurrenc0_.id=? for update
            nowait
Hibernate:
    update
        concurrency
    set
        version=?
    where
        id=?
        and version=?
```

```
thread-A] c.study.spa.service.ConcurrencyService    : 현재 count = 2
thread-A] c.study.spa.service.ConcurrencyService    : 변경 count = 3
```

```
Hibernate:
    update
        concurrency
    set
        count=?,
        version=?
    where
        id=?
        and version=?
```

위의 쿼리에서는 조회를 수행한 후 버전을 업데이트를 하고 count를 증가시키고 또 버전을 업데이트한다.

- 데이터를 수정할 때마다 버전 컬럼을 증가시킨다.
- 이를 통해 트랜잭션 간의 충돌을 방지하고 변경 내용을 추적할 수 있다.

락이 걸린 쿼리를 실행했을때만 버전 업데이트 쿼리가 추가된 것을 알 수 있었다.

3. **LockModeType.PESSIMISTIC_READ - 공유 잠금**

- 한 트랜잭션이 작업을 수행 중 일때 트랜잭션이 완료될때까지 다른 트랜잭션은 읽기만 수행 가능하다

```java
@Lock(LockModeType.PESSIMISTIC_READ)
@QueryHints({@QueryHint(name = "javax.persistence.lock.timeout", value = "1000")})
Optional<Concurrency> findById(Long id);
```

```
select
    concurrenc0_.id as id1_0_0_,
    concurrenc0_.count as count2_0_0_
from
    concurrency concurrenc0_
where
    concurrenc0_.id=? for share
```

```
Hibernate:
    update
        concurrency
    set
        count=?
    where
        id=?
```

```
thread-A] c.study.spa.service.ConcurrencyService   : 현재 count = 14
thread-A] c.study.spa.service.ConcurrencyService   : 변경 count = 15
```

```
thread-B] c.study.spa.service.ConcurrencyService   : 현재 count = 15
thread-B] c.study.spa.service.ConcurrencyService   : 변경 count = 16
```

for share

- 트랜잭션이 끝날때까지 select한 row값이 변경되지 않는 것을 보장한다.
- update 쿼리를 날리면 잠김 상태가 되고 트랜잭션이 끝날 때까지 대기한다.

애플리케이션 단에서 사용할 수 있는 동시성 제어 방법

ConcurrentHashMap

```java
 List<String> words = Arrays.asList("apple", "banana", "cat",

Map<Integer, List<String>> wordGroups = words.parallelStream(
        .collect(Collectors.groupingByConcurrent(
                            String::length,
```

```
                            ConcurrentHashMap::new, Collector
```

```
public V put(K key, V value) {
      return putVal(key, value, false);
}

 final V putVal(K key, V value, boolean onlyIfAbsent) {
      if (key == null || value == null) throw new NullPointer
      int hash = spread(key.hashCode());
      int binCount = 0;
      for (Node<K,V>[] tab = table;;) {
          Node<K,V> f; int n, i, fh; K fk; V fv;
          if (tab == null || (n = tab.length) == 0)
              tab = initTable();
          else if ((f = tabAt(tab, i = (n - 1) & hash)) == nu
              if (casTabAt(tab, i, null, new Node<K,V>(hash,
                  break;                   // no lock when ad
          }
          else if ((fh = f.hash) == MOVED)
              tab = helpTransfer(tab, f);
          else if (onlyIfAbsent // check first node without a
                   && fh == hash
                   && ((fk = f.key) == key || (fk != null &&
                   && (fv = f.val) != null)
              return fv;
          else {
              V oldVal = null;
              synchronized (f) {
                  if (tabAt(tab, i) == f) {
                      if (fh >= 0) {
                          binCount = 1;
                          for (Node<K,V> e = f;; ++binCount)
                              K ek;
                              if (e.hash == hash &&
                                  ((ek = e.key) == key ||
                                   (ek != null && key.equals(
                                  oldVal = e.val;
```

```
                                    if (!onlyIfAbsent)
                                        e.val = value;
                                    break;
                                }
                                Node<K,V> pred = e;
                                if ((e = e.next) == null) {
                                    pred.next = new Node<K,V>(h
                                    break;
                                }
                            }
                        }
                        else if (f instanceof TreeBin) {
                            Node<K,V> p;
                            binCount = 2;
                            if ((p = ((TreeBin<K,V>)f).putTreeV
                                                           valu
                                oldVal = p.val;
                                if (!onlyIfAbsent)
                                    p.val = value;
                            }
                        }
                        else if (f instanceof ReservationNode)
                            throw new IllegalStateException("Re
                    }
                }
                if (binCount != 0) {
                    if (binCount >= TREEIFY_THRESHOLD)
                        treeifyBin(tab, i);
                    if (oldVal != null)
                        return oldVal;
                    break;
                }
            }
        }
        addCount(1L, binCount);
        return null;
    }
```