


분산락 - 김민욱

■ 담당자	 민욱 김
■ 진행상황	시작 전
■ 최종 편집 일시	@2025년 6월 22일 오후 8:56

분산락이 뭘까?

분산락은 여러 대의 서버, 인스턴스, 스레드에서 공유 자원에 대한 동시 접근 제어를 하기 위한 락 메커니즘이다.

어떻게 구현할 수 있을까?

- Redis 기반
 - SET key value NX PX 3000 같은 식으로 raw한 명령어를 날린다.
 - NX ⇒ Map의 PutIfAbsent같은 기능인듯, 기존 키가 있다면 변경하지 않는다.
 - PX ⇒ 만료시간을 설정한다.
- MySQL의 Named Lock
 - GET_LOCK(str, timeout)
 - RELEASE_LOCK(str)
 - Spin Lock 기반

왜 Redis를 사용한 분산락이어야만 했을까?

- synchronized 기법
 - 서버 여러 대를 둘 예정이라 불가능했음
 - // TODO synchronized 동작 원리
- DB Named Lock
 - 커넥션 단위로 문자열에 대한 락을 획득한다. 즉, 락의 생명주기가 커넥션의 생명주기이다.

- 따라서, ~~Spring + JPA 에선 커넥션 풀을 사용하기 때문에 로직 중간에 커넥션이 바뀔 수 있음. 그렇게 되면 해당 락이 영구적으로 잡히버리는 문제가 발생한다.~~
- 락이 이미 점유되어 있다면, **Blocking Wait + 타임 아웃**을 사용한다.
 - wait 큐에 들어가고 sleep 상태로 변환 ⇒ CPU 점유 X
- TTL이 없어서, 락 해제에 실패한다면 답이 없다.
 - 우리처럼 땅 하나하나가 자원인 서비스에서는 쉽지 않은 접근이다.
- 또한 분산 DB 환경에서는 적용하지 못한다.
- // TODO Jpa, 커넥션, 네임드 락
- **DBMS에서 제공하는 비관적 락**
 - DBMS 레벨에서만 락이 적용된다.
 - **Row 단위로 잠금을 획득**하는 전통적인 RDB 락 전략.

`SELECT ... FOR UPDATE` 같은 구문으로 **해당 데이터에 대한 독점 접근 권한**을 획득한다.

`SELECT ... FOR UPDATE`이런 쿼리로 조회한번 때려서 락을 걸고, 수정을 해.
 - 락의 생명주기는 **트랜잭션의 생명주기**다. 즉, `commit` 또는 `rollback` 이 되기 전까지 절대 해제되지 않는다.
 - 우리의 땅따먹기 로직이 꽤나 무겁고 복잡해...
 - 만약에 도중에 어떤 컴포넌트가 멈추거나하면 해제가 불가능하지.
 - 락이 이미 점유되어 있다면, 대기자는 **Blocking Wait + 타임아웃** 구조로 기다리게 된다.
 - InnoDB 기준: `innodb_lock_wait_timeout` (기본 50초)
 - 락 보유자가 해제하면 그제서야 다음 대기자에게 락이 넘어간다.
 - 락 보유자가 예외나 실수로 커밋을 안 하면, 다른 요청은 **줄줄이 대기 or 데드락 발생**.
 - **TTL이 없다.** → 락은 **절대로 자동으로 풀리지 않는다**.
 - 락을 잡은 트랜잭션이 죽거나 강제 종료되지 않는 한, 락은 그대로 유지된다.
 - 분산 환경(서버 여러 대, DB 샤딩 등)에서 **락 공유가 불가능**하다.
 - 결국 단일 DB 스케일로 락 범위가 고정된다.
 - // TODO 분산 환경, 데드락, JPA와의 궁합
- **낙관적 락**

- 충돌이 '적을 것'이다 라고 가정하는 것
- 엔티티에 Version 필드를 두고, 업데이트하기 전에 조회하고 실제 업데이트할 때 버전을 확인한 후 결정한다.
- 재시도 로직이 없으면 그대로 실패처리 된다.
- 쿼리가 1+1로 나가고, 재시도 로직이 없으면 ⇒ 무조건 실패
 - 우리처럼 충돌이 잦은 땅이 분명한 서비스는 좀... 모토 자체가 맞지 않는다?

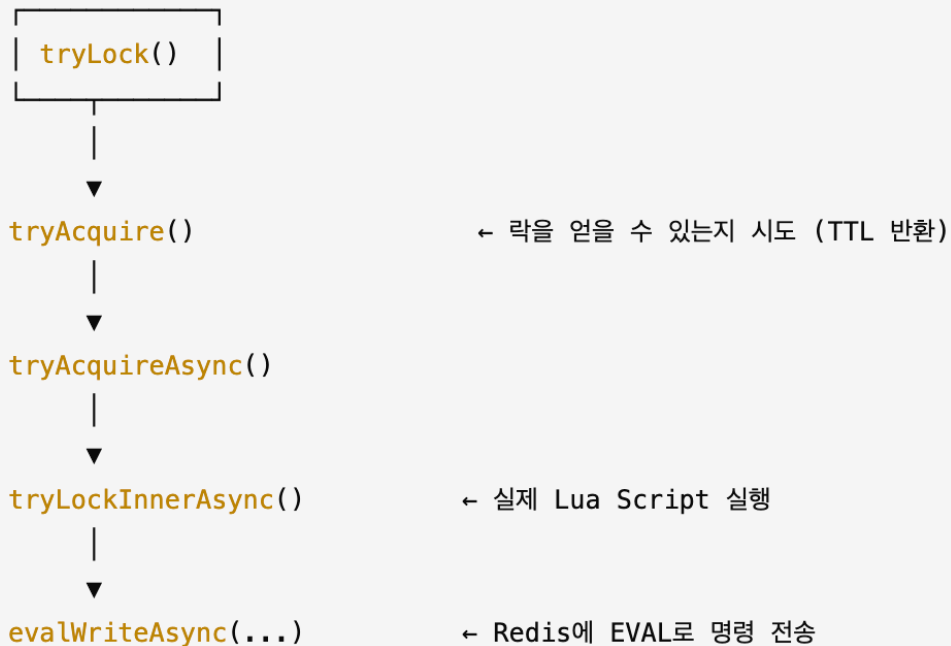
정리

네임드 락, 비관적 락은 TTL이 존재하지 않아서, 락 해제가 Fail 할 시 리스크가 매우 큼
 낙관적 락은 소수의 땅에서 집중적으로 경합이 일어나는 '그라운드 플립'의 특성 상 적합하지 않음.

Redis는 어떻게 원자적인 Lock 해제를 보장할까?

Redis 기반 분산락 (Redisson 기준)

- 락은 ****Redis key(Hash 구조)****를 기반으로 관리된다.
 - `lock:pixel:3:5` 같은 키에 락 소유자 정보를 저장한다.
- 락 소유자는 **"UUID:ThreadId"** 조합으로 구성된다.
 - UUID: RedissonClient 인스턴스 고유 ID
 - ThreadId: Java 쓰레드의 ID
 - 즉, JVM 단위 UUID + 스레드 단위 ID의 조합으로 **분산 환경에서도 유일성 보장**
- 락 해제는 **Lua Script**를 통해 원자적으로 처리된다.



- 조건 검사 (`hexists`), 카운트 감소 (`hincrby`), TTL 갱신 또는 삭제 (`pexpire` or `del`)까
지 한 번에 실행된다.
- 이 방식은 Redis의 **싱글 스레드 특성** 덕분에 완전한 원자성을 보장한다.
- TTL(lease time)을 지정할 수 있다.
 - Redis Key의 TTL로 할당이 되기 때문에, 레디스만 살아있으면 무조건 락 해제가 보
장
 - 락 보유자가 장애로 사라져도 자동으로 만료되어 **영구 락 상태 방지**
- 락 획득 실패 시에는 지정한 시간 (`waitTime`) 동안 대기 후 실패 처리된다.
- // TODO Lua script 직접 분석, tryLock 내부 구조 추적

분산 레디스 구조에선 어떻게 동작할까?

레디스를 여러 대 사용할 땐 크게 두 가지 방법이 있다.

센티넬 구조

- 1m + Ns (1개의 마스터, 3개 이상의 슬레이브(센티넬))
- Write 연산을 받으면 Replica들에게 비동기로 전송한다.

- 분산락 관점에서는 신경을 쓸 필요가 없어.
- KEY를 쓰고 읽는 거는 마스터에다가 요청 보내기 때문
- Master가 장애 시 Sentinel 중 하나를 Master로 승격

클러스터 구조

pgsql

복사 편집

6 노드 클러스터 (3 Master + 3 Replica)

[Node1] slots 0~5460 [Node4] = Replica of Node1

[Node2] slots 5461~10922 [Node5] = Replica of Node2

[Node3] slots 10923~16383 [Node6] = Replica of Node3

- 슬롯을 기준으로 여러 노드에 데이터를 분산 저장한다. (샤딩)



1. 원자성(Atomicity) 부족 — “멀티 키 연산이 깨진다”

Redis는 단일 노드에서는 다음과 같은 연산을 원자적으로 처리할 수 있어요:

```
bash
복사편집
MSET key1 val1 key2 val2
```

하지만 Cluster에선 키마다 다른 해시 슬롯에 매핑되기 때문에,
이런 멀티 키 연산이 아예 실패하거나 원자적으로 처리되지 않습니다.

```
bash
복사편집
MSET key1 val1 key2 val2 # 슬롯 다르면 오류!
```

- ✓ 같은 슬롯(=같은 노드)에 존재하는 키끼리는 가능
- ✗ 다른 슬롯이면 cross-slot error 발생

센티넬에서의 분산락

✓ 동작 방식

- Redisson은 Sentinel을 통해 **현재의 Master 주소**를 감지하고 그 노드에 락 명령을 보냄
- 클라이언트가 쓰기 연산을 마스터에만 보내므로 락도 마스터에 저장됨
- failover 발생 시, 새로운 마스터로 자동 전환

✓ 장점

- **고가용성 보장**: Redis 마스터 죽어도 락 기능 계속 유지 가능
- Redisson이 Sentinel 구성 자동 인식 (`redisson-sentinel.yaml`)

! 한계

- failover 중에는 락이 잠시 동안 유실될 수도 있음 (비동기 복제 지연)
- 락 해제 전 failover 발생하면 → 락 **orphan 현상** (소유자 불일치)

✓ 해결 전략

- Redisson은 락에 TTL을 두고 자동 만료하도록 하여 **orphan 락 문제를 완화**
- critical section을 idempotent하게 만들어야 함

클러스터에서의 분산락

✓ 동작 방식

- 락을 저장하는 key는 특정 슬롯(slot)에 매핑됨
- 해시 슬롯에 따라 락이 특정 노드에 분산 저장됨
- 락 관련 연산은 해당 슬롯을 가진 노드에서만 이루어짐

! 주의사항

- Redisson은 락 키가 단일 슬롯에 매핑되도록 강제해야 함
→ `lock:{pixel:x:y}` 이런 식으로 **해시태그**를 사용해야 함
- 그렇지 않으면 **CROSSSLOT** 오류 발생

! 한계

- 멀티 키 락 (예: 두 픽셀 락 잡기) 은 불가능하거나 매우 복잡해짐
- failover 시 일관성 깨질 수 있음 (레플리카가 최신 정보 갖고 있지 않으면)

✓ 장점

- 수평 확장 가능 → 락도 분산 처리 가능 (성능 이점)
- 노드 간 슬롯 재배포 지원

Redisson VS Lettuce

- Lettuce는 SET NX 같은 방식으로 락을 수동 구현해야한다.
- 해당 SET NX가 스핀락 방식이라 Redis에 부하 가능성
 - 정확히 말하자면, Lettuce는 Lock 자체를 지원하지 않는다.

- 단순히 RedisTemplate가 Redis와 통신을 하기 위한 '클라이언트' 역할을 해주는 것이다.
 - 실제 명령을 쏘는 역할을 Lettuce가 하는거지.
- 따라서 SET NX같은 명령어를 '직접' While문 같은 방식으로 Spin Lock으로서 구현을 해야하는 것이다.
- (참고) RedisTemplate는 Spring Data Redis에서 제공하는 API다.
 - ⇒ Spring에 의존성이 있다.
- 반면에 Redisson은 Spring과는 별도로 작동하며,
 - 직접 Redis에 커맨드를 날리고,
 - 고급 기능(분산락, 분산 캐시, RMap, RQueue 등)을 자체 구현합니다.

즉, RedissonClient.getLock() 이런 걸 쓰는 순간, **Spring의 RedisTemplate은 사용되지 않습니다.**

항목	Redisson	Lettuce
Lock 구현	✓ 내장된 RLock, FairLock, MultiLock 등 다양한 분산락 제공	✗ 직접 구현해야 함
원자성 보장	✓ Lua Script 기반의 락 해제/연장 원자 연산 제공	✗ 직접 구현해야 하며 실수 여지 있음
재진입 가능 락	✓ JVM 내 쓰레드 재진입 가능 (RLock)	✗ 직접 관리 필요
Watchdog (자동 연장)	✓ lockWatchdogTimeout으로 락 TTL 자동 연장	✗ 구현 필요
멀티 락 지원	✓ RedissonMultiLock, RedissonRedLock 등 제공	✗
구성 방식 지원	단일, Sentinel, Cluster 모두 지원	동일하게 지원
API 수준	고수준 API (RMap, RLock, RSet) 제공	Redis command 수준 API (저수준)
사용자 편의성	Spring 프로젝트에 통합하기 쉬움	기본적인 커맨드 수준만 제공

자바(스프링)에선 무슨 원리로 쓸 수 있는걸까?

▼ 실제 코드 + 주석

```
[Thread A] — subscribe() → RedissonLockEntry 생성 + pub/sub 구독 (1명)
```


[Thread B, C] — subscribe() → 이미 있는 Entry 사용 (구독 X)

[A, B, C] 모두 → RedissonLockEntry.getLatch().tryAcquire(...) 반복

→ unlock 발생 → pub/sub listener가 .latch.release() 호출

→ A, B, C 중 누군가 .tryAcquire() 성공 → 락 획득

```
public boolean tryLock(long waitTime, long leaseTime, TimeUnit unit) throws InterruptedException {
    long time = unit.toMillis(waitTime);
    long current = System.currentTimeMillis();
    long threadId = Thread.currentThread().getId();
    // 락 얻는 것을 시도한다. 만약 이미 락이 존재한다면 해당 락의 ttl을 반환한다.
    Long ttl = this.tryAcquire(waitTime, leaseTime, unit, threadId);
    if (ttl == null) {
        // 남은 ttl이 0 => 락이 없다 => 락 성공
        return true;
    } else {
        // 남은 wait time을 계산한다.
        time -= System.currentTimeMillis() - current;
        if (time <= 0L) {
            // 이미 wait time이 끝났다면 false 리턴
            this.acquireFailed(waitTime, unit, threadId);
            return false;
        } else {
            // 현재시간 다시 계산
            current = System.currentTimeMillis();
            // 구독을 시도한다. 구독도 내부적으로 세마포어를 사용하기 때문에 한 스레드만 구독이 가능하다.
            RFuture<RedissonLockEntry> subscribeFuture = this.subscribe(threadId);

            // 구독에 실패했다면 unsubscribe 한다.
            if (!subscribeFuture.await(time, TimeUnit.MILLISECONDS)) {
                if (!subscribeFuture.cancel(false)) {
```

```

        subscribeFuture.onComplete((res, e) → {
            if (e == null) {
                this.unsubscribe(subscribeFuture, threadId);
            }

        });
    }

    this.acquireFailed(waitTime, unit, threadId);
    return false;
} else {
    try {
        // wait time을 다시 계산한다.
        time -= System.currentTimeMillis() - current;
        if (time <= 0L) {
            this.acquireFailed(waitTime, unit, threadId);
            boolean var20 = false;
            return var20;
        } else {
            boolean var16;
            do {
                // 다시 락 시도
                long currentTime = System.currentTimeMillis();
                ttl = this.tryAcquire(waitTime, leaseTime, unit, threadId);

                // 락 성공
                if (ttl == null) {
                    var16 = true;
                    return var16;
                }

                // wait time 초과로 락 획득 실패
                time -= System.currentTimeMillis() - currentTime;
                if (time <= 0L) {
                    this.acquireFailed(waitTime, unit, threadId);
                    var16 = false;
                    return var16;
                }
            } while (true);
        }
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
        return false;
    }
}

```

```

        currentTime = System.currentTimeMillis();
        if (ttl >= 0L && ttl < time) {
            // wait time이 남았다면 ttl 전까지 재시도한다. 만약 락
            // 을 얻을 수 없다면 pub/sub 오기전까지 waitng한다.
            ((RedissonLockEntry)subscribeFuture.getNow()).ge
            tLatch().tryAcquire(ttl, TimeUnit.MILLISECONDS);
        } else {
            ((RedissonLockEntry)subscribeFuture.getNow()).ge
            tLatch().tryAcquire(time, TimeUnit.MILLISECONDS);
        }

        time -= System.currentTimeMillis() - currentTime;

        // wait time 끝날 때까지 반복
    } while(time > 0L);

    this.acquireFailed(waitTime, unit, threadId);
    var16 = false;
    return var16;
}
} finally {
    this.unsubscribe(subscribeFuture, threadId);
}
}
}
}
}

private Long tryAcquire(long waitTime, long leaseTime, TimeUnit unit, l
ong threadId) {
    return (Long)this.get(this.tryAcquireAsync(waitTime, leaseTime, uni
t, threadId));
}

private <T> RFuture<Long> tryAcquireAsync(long waitTime, long leas
eTime, TimeUnit unit, long threadId) {
    RFuture ttlRemainingFuture;
    if (leaseTime != -1L) {

```

```

        ttlRemainingFuture = this.tryLockInnerAsync(waitTime, leaseTime, unit, threadId, RedisCommands.EVAL_LONG);
    } else {
        ttlRemainingFuture = this.tryLockInnerAsync(waitTime, this.internalLockLeaseTime, TimeUnit.MILLISECONDS, threadId, RedisCommands.EVAL_LONG);
    }

    ttlRemainingFuture.onComplete((ttlRemaining, e) → {
        if (e == null) {
            if (ttlRemaining == null) {
                if (leaseTime != -1L) {
                    this.internalLockLeaseTime = unit.toMillis(leaseTime);
                } else {
                    this.scheduleExpirationRenewal(threadId);
                }
            }
        }
    });
    return ttlRemainingFuture;
}

```

// 실제 lua script를 통해 락을 얻는 부분

```

<T> RFuture<T> tryLockInnerAsync(long waitTime, long leaseTime, TimeUnit unit, long threadId, RedisStrictCommand<T> command) {
    return this.evalWriteAsync(this.getRawName(), LongCodec.INSTANCE, command, "if (redis.call('exists', KEYS[1]) == 0) then redis.call('hincrby', KEYS[1], ARGV[2], 1); redis.call('pexpire', KEYS[1], ARGV[1]); return nil; end; if (redis.call('hexists', KEYS[1], ARGV[2]) == 1) then redis.call('hincrby', KEYS[1], ARGV[2], 1); redis.call('pexpire', KEYS[1], ARGV[1]); return nil; end; return redis.call('pttl', KEYS[1]);", Collections.singletonList(this.getRawName()), new Object[]{unit.toMillis(leaseTime), this.getLockName(threadId)});
}

```

```

public class RedissonLockEntry implements PubSubEntry<RedissonLock

```

```

Entry> {
    private volatile int counter;
    private final Semaphore latch = new Semaphore(0);
    private final RPromise<RedissonLockEntry> promise;
    private final ConcurrentLinkedQueue<Runnable> listeners = new ConcurrentLinkedQueue();

    public RedissonLockEntry(RPromise<RedissonLockEntry> promise) {
        this.promise = promise;
    }

    public int acquired() {
        return this.counter;
    }

    public void acquire() {
        ++this.counter;
    }

    public int release() {
        return --this.counter;
    }

    public RPromise<RedissonLockEntry> getPromise() {
        return this.promise;
    }

    public void addListener(Runnable listener) {
        this.listeners.add(listener);
    }

    public boolean removeListener(Runnable listener) {
        return this.listeners.remove(listener);
    }

    public ConcurrentLinkedQueue<Runnable> getListeners() {
        return this.listeners;
    }
}

```

```

    public Semaphore getLatch() {
        return this.latch;
    }
}

    public RFuture<E> subscribe(String entryName, String channelName)
    {
        AsyncSemaphore semaphore = this.service.getSemaphore(new ChannelName(channelName));
        RPromise<E> newPromise = new RedissonPromise();
        semaphore.acquire(() → {
            if (!newPromise.setUncancellable()) {
                semaphore.release();
            } else {
                E entry = (PubSubEntry)this.entries.get(entryName);
                // 이미 엔트리가 있다면 구독하지 않고 해당 엔트리에 진입한다.
                if (entry != null) {
                    entry.acquire();
                    semaphore.release();
                    // 내부적으로 엔트리의 리스너에 등록
                    // 즉, 현재 entry의 promise가 끝난다면, 같이 결과를 전파받음
                    entry.getPromise().onComplete(new TransferListener(newPromise));
                } else {
                    // 엔트리가 없으면 새로 만든다.
                    E value = this.createEntry(newPromise);
                    value.acquire();

                    E oldValue = (PubSubEntry)this.entries.putIfAbsent(entryName, value);
                    // 이 사이에 새로 생겼는지 확인
                    if (oldValue != null) {
                        oldValue.acquire();
                        semaphore.release();
                        oldValue.getPromise().onComplete(new TransferListener(newPromise));
                    } else {
                        // 진짜 새로 생긴거라면 '구독'한다.
                        RedisPubSubListener<Object> listener = this.createListen

```

```

er(channelName, value);
        this.service.subscribe(LongCodec.INSTANCE, channelName, semaphore, new RedisPubSubListener[]{listener});
    }
}
});
return new Promise;
}

```

락을 얻는데 실패하면 우리는 어떻게 할 수 있을까?

1. 단순 실패처리
2. 재시도 로직 (지수 backoff)

```

int retry = 3;
boolean locked = false;
while (retry-- > 0) {
    locked = lock.tryLock(1, 5, TimeUnit.SECONDS);
    if (locked) break;
    Thread.sleep(100); // 또는 Exponential Backoff
}
if (!locked) {
    throw new CustomRetryFailException();
}

```

```

public void occupyPixelWithRetry(PixelOccupyRequest pixelOccupyRequest) {
    String lockName = REDISSON_LOCK_PREFIX + pixelOccupyRequest.getX() + pixelOccupyRequest.getY();
    RLock rLock = redissonClient.getLock(lockName);

    int maxRetry = 5;
    int baseDelayMillis = 200;
    long waitTime = 5L;
    long leaseTime = 3L;

```

```

    TimeUnit timeUnit = TimeUnit.SECONDS;

    for (int retry = 0; retry < maxRetry; retry++) {
        try {
            boolean available = rLock.tryLock(waitTime, leaseTime, timeUnit);
            if (available) {
                try {
                    pixelManager.occupyPixel(pixelOccupyRequest);
                    return; // 성공 시 메서드 종료
                } finally {
                    if (rLock.isHeldByCurrentThread()) {
                        rLock.unlock();
                    }
                }
            }
        }

        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            throw new RuntimeException("Thread interrupted while trying to
acquire lock", e);
        }

        // ! 락 획득 실패 → 백오프 후 재시도
        try {
            long backoff = baseDelayMillis * (1L << retry); // Exponential Back
off: 200, 400, 800, ...
            Thread.sleep(backoff + ThreadLocalRandom.current().nextInt(10
0)); // Jitter 추가
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            throw new RuntimeException("Interrupted during backoff sleep",
e);
        }
    }

    // 재시도 후에도 실패 → 예외 처리
    throw new AppException(ErrorCode.LOCK_ACQUISITION_ERROR);
}

```


<https://mangkyu.tistory.com/311>

<https://coding-review.tistory.com/542>

<https://jungguji.github.io/2025/01/09/%EB%8F%99%EC%8B%9C%EC%84%B1-%EC%A0%9C%EC%96%B4%EB%A5%BC-%EC%9C%84%ED%95%9C-Redisson-tryLock-%EB%A9%94%EC%84%9C%EB%93%9C%EC%9D%98-%EC%9E%91%EB%8F%99-%EC%9B%90%EB%A6%AC/>