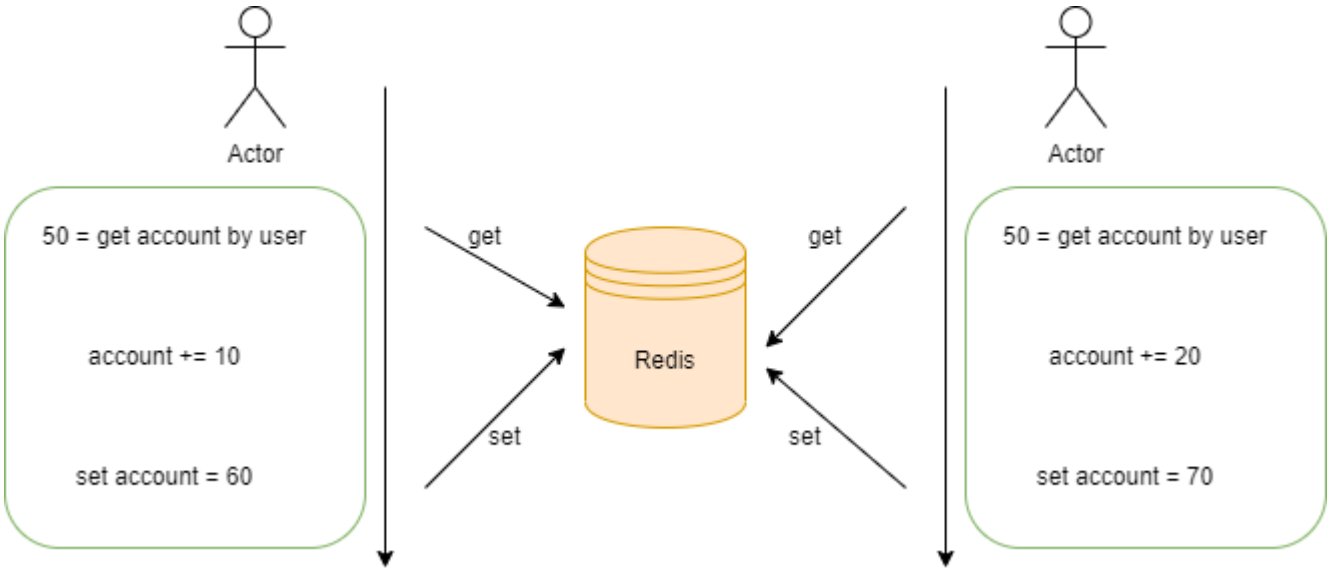
消除竞态之分布式锁

一、什么是竞态

> 竞争冒险（race hazard）又名竞态条件、竞争条件（race condition），它旨在描述一个系统或者进程的输出依赖于不受控制的事件出现顺序或者出现时机。此词源自于两个信号试着彼此竞争，来影响谁先输出。
> 举例来说，如果计算机中的两个进程同时试图修改一个共享内存的内容，在没有并发控制的情况下，最后的结果依赖于两个进程的执行顺序与时机。而且如果发生了并发访问冲突，则最后的结果是不正确的。
> 竞争冒险常见于不良设计的电子系统，尤其是逻辑电路。但它们在软件中也比较常见，尤其是有采用多线程技术的软件。

1.1 举个例子



二、分布式锁的实现

2.1 相关Redis命令

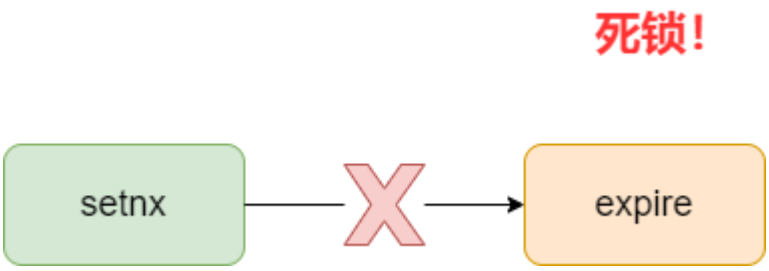
- setnx（set if not exist）
- expire
- del

2.2 Java实现

```
if (jedis.setnx("lock", "1") == 1) {
    try {
        System.out.println(Thread.currentThread().getId() + ":抢到了锁，可以执行业务");
        jedis.expire("lock", 3);// 可能死锁
        Thread.sleep(1000);
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        jedis.del("lock");
    }
} else {
    System.out.println(Thread.currentThread().getId() + ":没抢到锁");
}
```

存在问题

- setnx和expire之间不具备原子性，可能出现死锁



解决方案

在Redis 2.8版本增加setnx和expire组合在一起的原子指令**setex**

```
if ("OK".equals(jedis.set("lock","1",new SetParams().nx().ex(3L)))) {
    try {
        System.out.println(Thread.currentThread().getId() + ":抢到了锁，可以执行业务");
        Thread.sleep(6000);
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        jedis.del("lock");
    }
} else {
```

```
    System.out.println(Thread.currentThread().getId() + ":没抢到锁");
}
```

2.3 超时问题

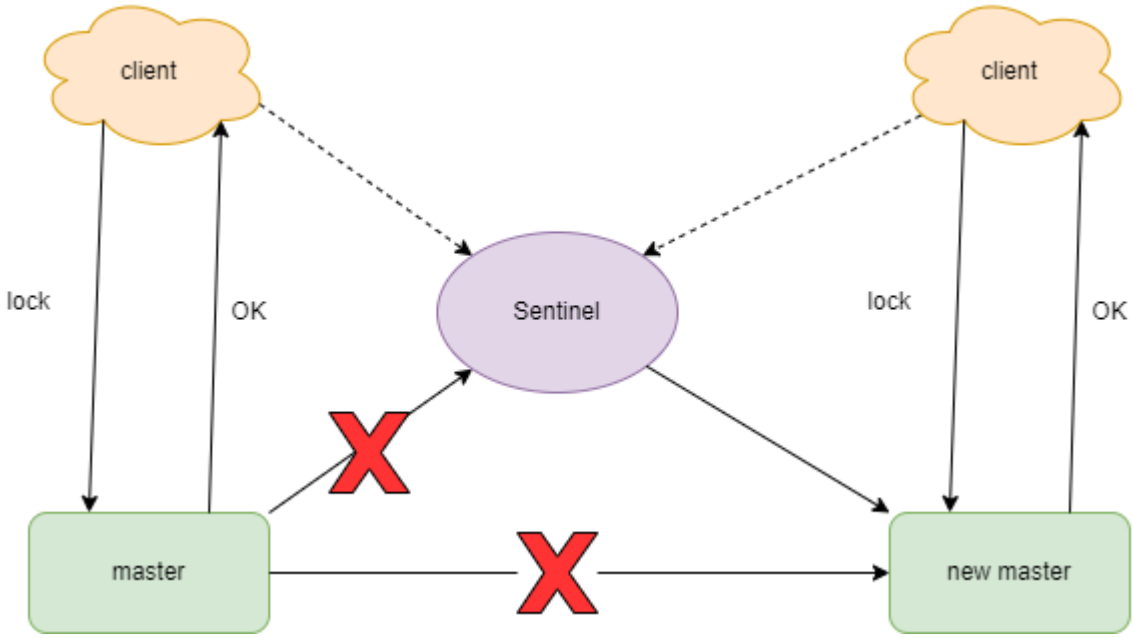
1. 业务未执行完，锁的超时时间已到。导致锁被释放后，存在并发执行问题。
2. 在1问题的基础上，将其他线程的锁释放了，导致更加的混乱。

因此，不要将Redis分布锁用于长时间的事务。
另外可以将锁的value设置成一个随机数，在释放之前比较随机数，这样子就避免了释放其他线程的锁的问题。

Java实现

```
String uuid = UUID.randomUUID().toString();
if ("OK".equals(jedis.set("lock", uuid, new SetParams().nx().ex(1L)))) {
    try {
        System.out.println(Thread.currentThread().getId() + ":抢到了锁，可以执行业务");
        Thread.sleep(2000);
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        if (uuid.equals(jedis.get("lock"))) {  // 不具备原子性
            jedis.del("lock");
        }
    }
} else {
    System.out.println(Thread.currentThread().getId() + ":没抢到锁");
}
```

2.4 集群问题



在集群中，当主节点挂掉时，从节点会取而代之，但是客户端并没有感知。这时候如果原先一个客户端在主节点申请到了一把锁，但是还没来得及同步到从节点时突然挂掉，这时候从节点升级成了主节点，但是并没有之前那把锁，这时候又有一个客户端来申请锁，会申请成功。这时候就会出现不安全的情况了。

三、扩展之Redlock算法

3.1 Redisson

Redisson Github 首页

README.md

Redisson - Redis Java client with features of an in-memory data grid

maven central 3.16.8 | javadoc 3.16.8 | license apache

Quick start | Documentation | Changelog | Code examples | FAQs | Report an issue

Based on high-performance async and lock-free Java Redis client and Netty framework.
JDK compatibility: 1.8 - 17, Android

3.2 TryLock

Java实现

```
Config config = new Config();
config.useSingleServer().setAddress("redis://127.0.0.1:6379").setPassword("
123456");
RedissonClient client = Redisson.create(config);
RLock lock = client.getLock("lock");
try {
    boolean res = lock.tryLock(2L, 4L, TimeUnit.SECONDS);
    if (res) {
        // 拿到锁，执行业务
 System.out.println(Thread.currentThread().getId() + ": 拿到了锁");
        Thread.sleep(3000);
    } else {
        System.out.println(Thread.currentThread().getId() + ": 没拿到锁");
    }
} catch (Exception ex) {
    ex.printStackTrace();
} finally {
    lock.unlock();
}
```

tryLock源码

```
// 三个参数分别是等待时间，锁过期时间，时间单位
public boolean tryLock(long waitTime, long leaseTime, TimeUnit unit) throws
InterruptedException {
        long time = unit.toMillis(waitTime);  // 取得最大等待时间
        long current = System.currentTimeMillis();  // 当前时间
        final long threadId = Thread.currentThread().getId(); // 当前线程ID
 (实现可重入锁)
        Long ttl = tryAcquire(leaseTime, unit, threadId);  // 尝试获取锁，会
返回一个过期时间
        // lock acquired
 if (ttl == null) {  // 没有过期时间，表示获取锁成功
            return true;
        }

        time -= (System.currentTimeMillis() - current);
        if (time <= 0) {  // 表示过期时间大于等待时间，直接获取失败
            acquireFailed(threadId);
```

```
            return false;
        }

        current = System.currentTimeMillis();
        final RFuture<RedissonLockEntry> subscribeFuture =
subscribe(threadId);  // 订阅锁释放事件消息
        if (!await(subscribeFuture, time, TimeUnit.MILLISECONDS)) {
            if (!subscribeFuture.cancel(false)) {
                subscribeFuture.addListener(new
FutureListener<RedissonLockEntry>() {
                    @Override
 public void operationComplete(Future<RedissonLockEntry> future) throws
Exception {
                        if (subscribeFuture.isSuccess()) {  // 监听解锁消
息，释放许可
                            unsubscribe(subscribeFuture, threadId);
                        }
                    }
                });
            }
            acquireFailed(threadId);
            return false;
        }

        try {
            time -= (System.currentTimeMillis() - current);
            if (time <= 0) {  // 等待是否超时
                acquireFailed(threadId);
                return false;
            }

            while (true) {
                long currentTime = System.currentTimeMillis();
                ttl = tryAcquire(leaseTime, unit, threadId);  // 尝试获取锁
                // lock acquired
 if (ttl == null) {
                    return true;
                }
```

```
                time -= (System.currentTimeMillis() - currentTime);
                if (time <= 0) {
                    acquireFailed(threadId);
                    return false;
                }

                // waiting for message
                                    // 通过信号量阻塞等待锁释放的消息
 currentTime = System.currentTimeMillis();
                if (ttl >= 0 && ttl < time) {
                    getEntry(threadId).getLatch().tryAcquire(ttl,
TimeUnit.MILLISECONDS);
                } else {
                    getEntry(threadId).getLatch().tryAcquire(time,
TimeUnit.MILLISECONDS);
                }

                time -= (System.currentTimeMillis() - currentTime);
                if (time <= 0) {
                    acquireFailed(threadId);
                    return false;
                }
            }
        } finally {
            unsubscribe(subscribeFuture, threadId);  // 取消订阅
        }
//        return get(tryLockAsync(waitTime, leaseTime, unit));
 }
```

tryAcquire源码

```
<T> RFuture<T> tryLockInnerAsync(long leaseTime, TimeUnit unit, long
threadId, RedisStrictCommand<T> command) {
    internalLockLeaseTime = unit.toMillis(leaseTime);

    return commandExecutor.evalWriteAsync(getName(), LongCodec.INSTANCE,
command,
              "if (redis.call('exists', KEYS[1]) == 0) then " +  // 判断key
释放存在
```

```
                    "redis.call('hset', KEYS[1], ARGV[2], 1); " +  // 不存在
就将key放入Hash结构，field为"UUID:线程ID"
                    "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); " +  // 锁被
占据了，如果是当前线程占据的，计数加一，可以重入
                    "redis.call('pexpire', KEYS[1], ARGV[1]); " +
                    "return nil; " +
                "end; " +
                "return redis.call('pttl', KEYS[1]);",  // 不是当前线程的锁，
返回过期时间
                  Collections.<Object>singletonList(getName()),
internalLockLeaseTime, getLockName(threadId));
    }
```

3.3 额外拓展RedLock

解决集群模式下的Redis分布式锁的漏洞。
依次从多个节点获取锁，只有N/2+1节点的成功，才算成功获取到锁。