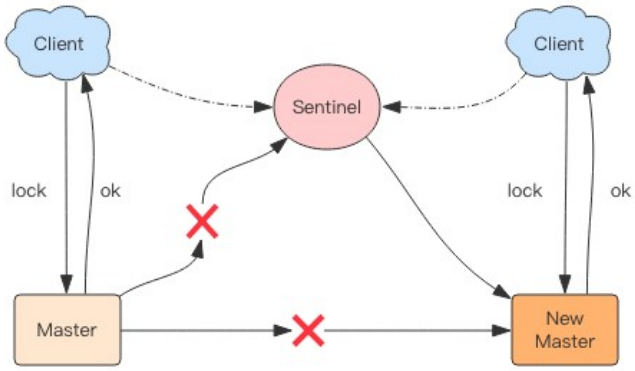
拓展 3：拾遗漏补 —— 再谈分布式锁

在第三节，我们细致讲解了分布式锁的原理，它的使用非常简单，一条指令就可以完成加锁操作。不过在集群环境下，这种方式是有缺陷的，它不是绝对安全的。

比如在 Sentinel 集群中，主节点挂掉时，从节点会取而代之，客户端上却并没有明显感知。原先第一个客户端在主节点中申请成功了一把锁，但是这把锁还没有来得及同步到从节点，主节点突然挂掉了。然后从节点变成了主节点，这个新的节点内部没有这个锁，所以当另一个客户端过来请求加锁时，立即就批准了。这样就会导致系统中同样一把锁被两个客户端同时持有，不安全性由此产生。



不过这种不安全也仅仅是在主从发生 failover 的情况下才会产生，而且持续时间极短，业务系统多数情况下可以容忍。

Redlock 算法

为了解决这个问题，Antirez 发明了 Redlock 算法，它的流程比较复杂，不过已经有了很多开源的 library 做了良好的封装，用户可以拿来即用，比如 redlock-py。

```py
import redlock

addrs = [{
    "host": "localhost",
    "port": 6379,
    "db": 0
}, {
    "host": "localhost",
    "port": 6479,
    "db": 0
}, {
    "host": "localhost",
    "port": 6579,
    "db": 0
}]
dlm = redlock.Redlock(addrs)
success = dlm.lock("user-lck-laoqian", 5000)
if success:
    print 'lock success'
    dlm.unlock('user-lck-laoqian')
else:
    print 'lock failed'
```

为了使用 Redlock，需要提供多个 Redis 实例，这些实例之前相互独立没有主从关系。同很多分布式算法一样，redlock 也使用「大多数机制」。

加锁时，它会向过半节点发送 `set(key, value, nx=True, ex=xxx)` 指令，只要过半节点 `set` 成功，那就认为加锁成功。释放锁时，需要向所有节点发送 `del` 指令。不过 Redlock 算法还需要考虑出错重试、时钟漂移等很多细节问题，同时因为 Redlock 需要向多个节点进行读写，意味着相比单实例 Redis 性能会下降一些。

Redlock 使用场景

如果你很在乎高可用性，希望挂了一台 redis 完全不受影响，那就应该考虑 redlock。不过代价也是有的，需要更多的 redis 实例，性能也下降了，代码上还需要引入额外的 library，运维上也需要特殊对待，这些都是需要考虑的成本，使用前请再三斟酌。

扩展阅读

1. 你以为 Redlock 算法真的很完美?

2. Redlock-py 的作者其人趣事

看头像非常酷，这位老哥的 Github 地址是 github.com/optimuspaul