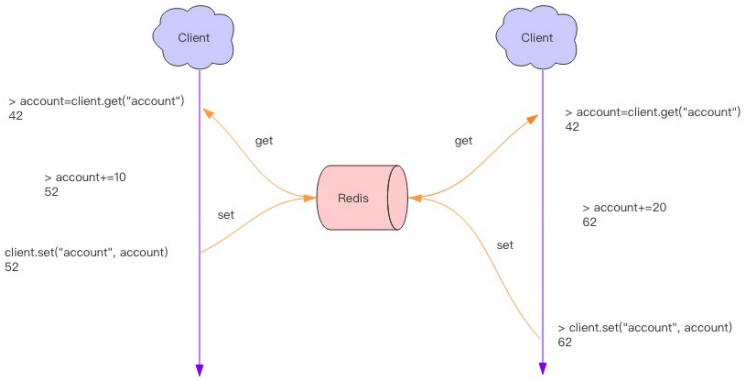
应用 1：千帆竞发 —— 分布式锁

分布式应用进行逻辑处理时经常会遇到并发问题。

比如一个操作要修改用户的状态，修改状态需要先读出用户的状态，在内存里进行修改，改完了再存回去。如果这样的操作同时进行了，就会出现并发问题，因为读取和保存状态这两个操作不是原子的。（Wiki 解释：所谓**原子操作**是指不会被线程调度机制打断的操作；这种操作一旦开始，就一直运行到结束，中间不会有任何 context switch 线程切换。）



这个时候就要使用到分布式锁来限制程序的并发执行。Redis 分布式锁使用非常广泛，它是面试的重要考点之一，很多同学都知道这个知识，也大致知道分布式锁的原理，但是具体到细节的使用上往往并不完全正确。

分布式锁

分布式锁本质上要实现的目标就是在 Redis 里面占一个“茅坑”，当别的进程也要来占时，发现已经有人蹲在那里了，就只好放弃或者稍后再试。

占坑一般是使用 setnx(set if not exists) 指令，只允许被一个客户端占坑。先来先占， 用完了，再调用 del 指令释放茅坑。

```
// 这里的冒号:就是一个普通的字符，没特别含义，它可以是任意其它字符，不要误解
> setnx lock:codehole true
OK
... do something critical ...
> del lock:codehole
(integer) 1
```

但是有个问题，如果逻辑执行到中间出现异常了，可能会导致 del 指令没有被调用，这样就会陷入死锁，锁永远得不到释放。

于是我们在拿到锁之后，再给锁加上一个过期时间，比如 5s，这样即使中间出现异常也可以保证 5 秒之后锁会自动释放。

```
> setnx lock:codehole true
OK
> expire lock:codehole 5
... do something critical ...
> del lock:codehole
(integer) 1
```

但是以上逻辑还有问题。如果在 setnx 和 expire 之间服务器进程突然挂掉了，可能是因为机器掉电或者是被人为杀掉的，就会导致 expire 得不到执行，也会造成死锁。

这种问题的根源就在于 setnx 和 expire 是两条指令而不是原子指令。如果这两条指令可以一起执行就不会出现问题。也许你会想到用 Redis 事务来解决。但是这里不行，因为 expire 是依赖于 setnx 的执行结果的，如果 setnx 没抢到锁，expire 是不应该执行的。事务里没有 if-else 分支逻辑，事务的特点是一口气执行，要么全部执行要么一个都不执行。

为了解决这个疑难，Redis 开源社区涌现了一堆分布式锁的 library，专门用来解决这个问题。实现方法极为复杂，小白用户一般要费很大的精力才可以搞懂。如果你需要使用分布式锁，意味着你不能仅仅使用 Jedis 或者 redis-py 就行了，还得引入分布式锁的 library。

为了治理这个乱象，Redis 2.8 版本中作者加入了 set 指令的扩展参数，使得 setnx 和 expire 指令可以一起执行，彻底解决了分布式锁的乱象。从此以后所有的第三方分布式锁 library 可以休息了。

```
> set lock:codehole true ex 5 nx
OK
... do something critical ...
> del lock:codehole
```

上面这个指令就是 setnx 和 expire 组合在一起的原子指令，它就是分布式锁的奥义所在。

超时问题

Redis 的分布式锁不能解决超时问题，如果在加锁和释放锁之间的逻辑执行的太长，以至于超出了锁的超时限制，就会出现问题。因为这时候锁过期了，第二个线程重新持有了这把锁，但是紧接着第一个线程执行完了业务逻辑，就把锁给释放了，第三个线程就会在第二个线程逻辑执行完之间拿到了锁。

为了避免这个问题，Redis 分布式锁不要用于较长时间的任务。如果真的偶尔出现了，数据出现的小波错乱可能需要人工介入解决。

py

```
tag = random.nextint()  # 随机数
if redis.set(key, tag, nx=True, ex=5):
    do_something()
    redis.delifequals(key, tag)  # 假象的 delifequals 指令
```

有一个更加安全的方案是为 set 指令的 value 参数设置为一个随机数，释放锁时先匹配随机数是否一致，然后再删除 key。但是匹配 value 和删除 key 不是一个原子操作，Redis 也没有提供类似于 `delifequals` 这样的指令，这就需要使用 Lua 脚本来处理了，因为 Lua 脚本可以保证连续多个指令的原子性执行。

```lua
# delifequals
if redis.call("get",KEYS[1]) == ARGV[1] then
    return redis.call("del",KEYS[1])
else
    return 0
end
```

可重入性

可重入性是指线程在持有锁的情况下再次请求加锁，如果一个锁支持同一个线程的多次加锁，那么这个锁就是可重入的。比如 Java 语言里有个 ReentrantLock 就是可重入锁。Redis 分布式锁如果要支持可重入，需要对客户端的 set 方法进行包装，使用线程的 Threadlocal 变量存储当前持有锁的计数。

```
# -*- coding: utf-8
import redis
import threading


locks = threading.local()
locks.redis = {}

def key_for(user_id):
    return "account_{}".format(user_id)

def _lock(client, key):
    return bool(client.set(key, True, nx=True, ex=5))

def _unlock(client, key):
    client.delete(key)

def lock(client, user_id):
    key = key_for(user_id)
    if key in locks.redis:
        locks.redis[key] += 1
        return True
    ok = _lock(client, key)
```

```
        if not ok:
            return False
        locks.redis[key] = 1
        return True

    def unlock(client, user_id):
        key = key_for(user_id)
        if key in locks.redis:
            locks.redis[key] -= 1
            if locks.redis[key] <= 0:
                del locks.redis[key]
            return True
        return False

    client = redis.StrictRedis()
    print "lock", lock(client, "codehole")
    print "lock", lock(client, "codehole")
    print "unlock", unlock(client, "codehole")
    print "unlock", unlock(client, "codehole")
```

以上还不是可重入锁的全部，精确一点还需要考虑内存锁计数的过期时间，代码复杂度将会继续升高。老钱不推荐使用可重入锁，它加重了客户端的复杂性，在编写业务方法时注意在逻辑结构上进行调整完全可以不使用可重入锁。下面是 Java 版本的可重入锁。

```
public class RedisWithReentrantLock {

  private ThreadLocal<Map<String, Integer>> lockers = new ThreadLocal<>();

  private Jedis jedis;

  public RedisWithReentrantLock(Jedis jedis) {
    this.jedis = jedis;
  }

  private boolean _lock(String key) {
    return jedis.set(key, "", "nx", "ex", 5L) != null;
  }

  private void _unlock(String key) {
    jedis.del(key);
  }

  private Map<String, Integer> currentLockers() {
    Map<String, Integer> refs = lockers.get();
    if (refs != null) {
```

```
      return refs;
    }
    lockers.set(new HashMap<>());
    return lockers.get();
  }

  public boolean lock(String key) {
    Map<String, Integer> refs = currentLockers();
    Integer refCnt = refs.get(key);
    if (refCnt != null) {
      refs.put(key, refCnt + 1);
      return true;
    }
    boolean ok = this._lock(key);
    if (!ok) {
      return false;
    }
    refs.put(key, 1);
    return true;
  }

  public boolean unlock(String key) {
    Map<String, Integer> refs = currentLockers();
    Integer refCnt = refs.get(key);
    if (refCnt == null) {
      return false;
    }
    refCnt -= 1;
    if (refCnt > 0) {
      refs.put(key, refCnt);
    } else {
      refs.remove(key);
      this._unlock(key);
    }
    return true;
  }

  public static void main(String[] args) {
    Jedis jedis = new Jedis();
    RedisWithReentrantLock redis = new RedisWithReentrantLock(jedis);
    System.out.println(redis.lock("codehole"));
    System.out.println(redis.lock("codehole"));
    System.out.println(redis.unlock("codehole"));
    System.out.println(redis.unlock("codehole"));
  }

}
```

跟 Python 版本区别不大，也是基于 ThreadLocal 和引用计数。

以上还不是分布式锁的全部，在小册的拓展篇《拾遗漏补 —— 再谈分布式锁》，我们还会继续对分布式锁做进一步的深入理解。

思考题

1. Review 下你自己的项目代码中的分布式锁，它的使用方式是否标准正确？
2. 如果你还没用过分布式锁，想想自己的项目中是否可以用上？