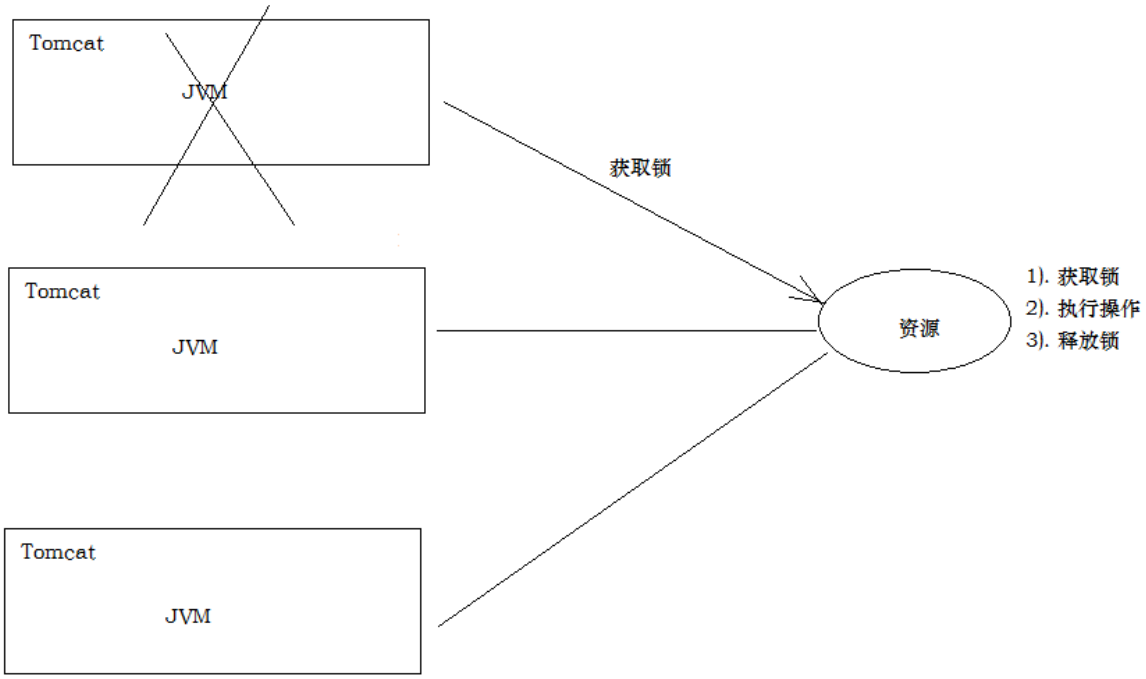
分布式锁

1.1 什么是分布式锁

- 当在分布式模型下，数据只有一份（或有限制），此时需要利用锁的技术控制某一时刻修改数据的进程数。
- 与单机模式下的锁不仅需要保证进程可见，还需要考虑进程与锁之间的网络问题。（我觉得分布式情况下之所以问题变得复杂，主要就是需要考虑到网络的延时和不可靠。）
- 分布式锁还是可以将标记存在内存，只是该内存不是某个进程分配的内存而是公共内存如 Redis、Memcache。至于利用数据库、文件等做锁与单机的实现是一样的，只要保证标记能互斥就行。

1.2 分布式锁场景



1.3 Redis实现分布式锁

```
public class RedisTool {

    private static final String LOCK_SUCCESS = "OK";

    /**
     * 尝试获取分布式锁
     * @param jedis Redis客户端
     * @param lockKey 锁
     * @param requestId 请求标识
     * @param expireTime 超期时间
     * @return 是否获取成功
     */
    public static boolean tryGetDistributedLock(Jedis jedis, String lockKey,
String requestId, int expireTime) {

        SetParams params = new SetParams();
```

```
            params = params.nx();
            params = params.ex(expireTime);
            String result = jedis.set(lockKey, requestId, params);

            if (LOCK_SUCCESS.equals(result)) {
                return true;
            }
            return false;
        }


        /**
         * 释放锁
         * @param jedis
         * @param lockKey
         */
        public static void releaseLock(Jedis jedis, String lockKey) {
            jedis.del(lockKey);
        }

    }
```

可以看到，我们加锁就一行代码：jedis.set(lockKey, requestId, params)，这个set()方法一共有三个形参：

1. 第一个为key，我们使用key来当锁，因为key是唯一的。
2. 第二个为value，我们传的是requestId，很多童鞋可能不明白，有key作为锁不就够了吗，为什么还要用到value？原因就是我们在上面讲到可靠性时，分布式锁要满足第四个条件解铃还须系铃人，通过给value赋值为requestId，我们就知道这把锁是哪个请求加的了，在解锁的时候就可以有依据。requestId可以使用UUID.randomUUID().toString()方法生成。
3. 第三个为SetParams，这个参数我们调用了 nx , ex ;
4. 第四个为nx，意思是 SET IF NOT EXIST，即当key不存在时，我们进行set操作；若key已经存在，则不做任何操作；
5. 第五个为ex，代表key的过期时间;

总的来说，执行上面的set()方法就只会导致两种结果：

1. 当前没有锁（key不存在），那么就进行加锁操作，并对锁设置个有效期，同时value表示加锁的客户端。
2. 已有锁存在，不做任何操作。

心细的童鞋就会发现了，我们的加锁代码满足我们可靠性里描述的三个条件:

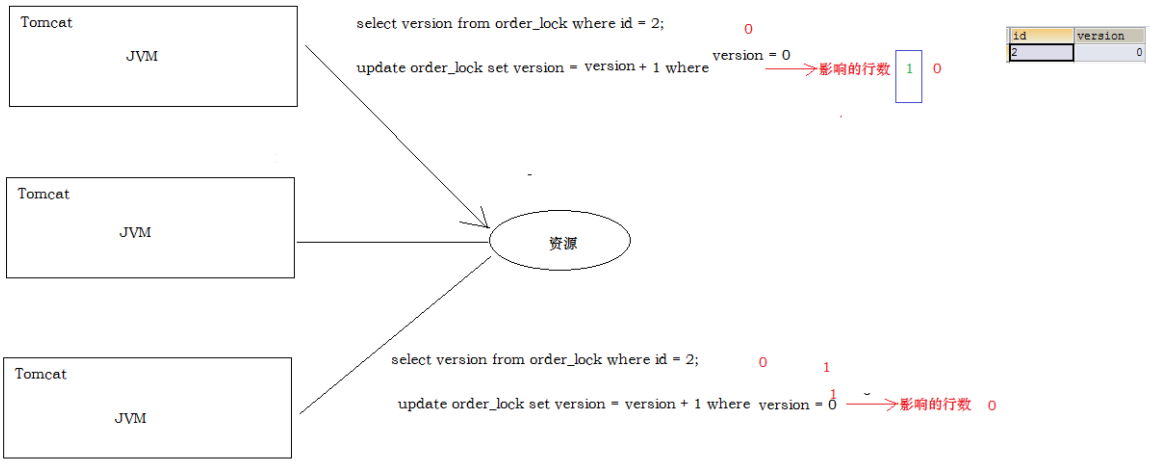
首先，set()加入了NX参数，可以保证如果已有key存在，则函数不会调用成功，也就是只有一个客户端能持有锁，满足互斥性。

其次，由于我们对锁设置了过期时间，即使锁的持有者后续发生崩溃而没有解锁，锁也会因为到了过期时间而自动解锁（即key被删除），不会发生死锁。

最后，因为我们将value赋值为requestId，代表加锁的客户端请求标识，那么在客户端在解锁的时候就可以进行校验是否是同一个客户端。由于我们只考虑Redis单机部署的场景，所以容错性我们暂不考虑。

1.4 MySQL 实现分布式锁

可以基于MySQL的乐观锁来实现分布式锁；



1.5 Zookeeper 实现分布式锁

1). 定义接口

```
public interface ExtLock {
    // 获取锁
    public void getLock();

    // 释放锁
    public void unLock();
}
```

2). 定义抽象类，采用模板方法涉及模式

```
public abstract class ZookeeperAbstractLock implements ExtLock {

    // 集群连接地址
    protected String CONNECTION = "127.0.0.1:2181";
    // zk客户端连接
    protected ZkClient zkClient = new ZkClient(CONNECTION);
    // path路径
    protected String lockPath = "/path";
    protected CountDownLatch countDownLatch = new CountDownLatch(1);

    public void getLock() {
        if (tryLock()) {
            System.out.println("####获取锁成功######");
        } else {
            waitLock();
            getLock();
        }
    }

    // 获取锁
    abstract boolean tryLock();

    // 等待锁
```

```
    abstract void waitLock();

    public void unLock() {
        if (zkClient != null) {
            System.out.println("########释放锁##########");
            zkClient.close();
        }
    }
}
```

3). 具体的Zookeeper分布式锁实现

```
public class ZookeeperDistrbuteLock extends ZookeeperAbstractLock {

    @Override
    boolean tryLock() {
        try {
            zkClient.createEphemeral(lockPath);
            return true;
        } catch (Exception e) {
            return false;
        }

    }

    @Override
    void waitLock() {

        // 使用zk临时事件监听
        IZkDataListener iZkDataListener = new IZkDataListener() {

            public void handleDataDeleted(String path) throws Exception {
                if (countDownLatch != null) {
                    countDownLatch.countDown();
                }
            }

            public void handleDataChange(String arg0, Object arg1) throws
Exception {

            }
        };
        // 注册事件通知
        zkClient.subscribeDataChanges(lockPath, iZkDataListener);
        if (zkClient.exists(lockPath)) {
            countDownLatch = new CountDownLatch(1);
            try {
                countDownLatch.await();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
        // 监听完毕后，移除事件通知
        zkClient.unsubscribeDataChanges(lockPath, iZkDataListener);
    }

}
```

4). 测试

```
public class OrderServiceWithLock implements Runnable {

    private OrderNumGenerator orderNumGenerator = new OrderNumGenerator();
    private ExtLock extLock = new ZookeeperDistrbuteLock();

    public void run() {
        getNumber();
    }

    public void getNumber() {
        try {
            extLock.getLock();
            String number = orderNumGenerator.getNumber();
            System.out.println("线程:" + Thread.currentThread().getName() + ",生
成订单id:" + number);
        } catch (Exception e) {

        } finally {
            extLock.unLock();
        }
    }

    public static void main(String[] args) {
        System.out.println("多线程生成number");

        for (int i = 0; i < 100; i++) {
            new Thread(new OrderServiceWithLock()).start();
        }
    }

}
```

```
public class OrderNumGenerator {

    // 生成订单号规则
    private static int count = 0;

    public String getNumber() {
        try {
        //  Thread.sleep(200);
        } catch (Exception e) {
            e.printStackTrace();
        }
        SimpleDateFormat simpt = new SimpleDateFormat("yyyyMMddHHmmss");
        return simpt.format(new Date()) + "-" + ++count;
    }

}
```