解析分布式锁之redis实现

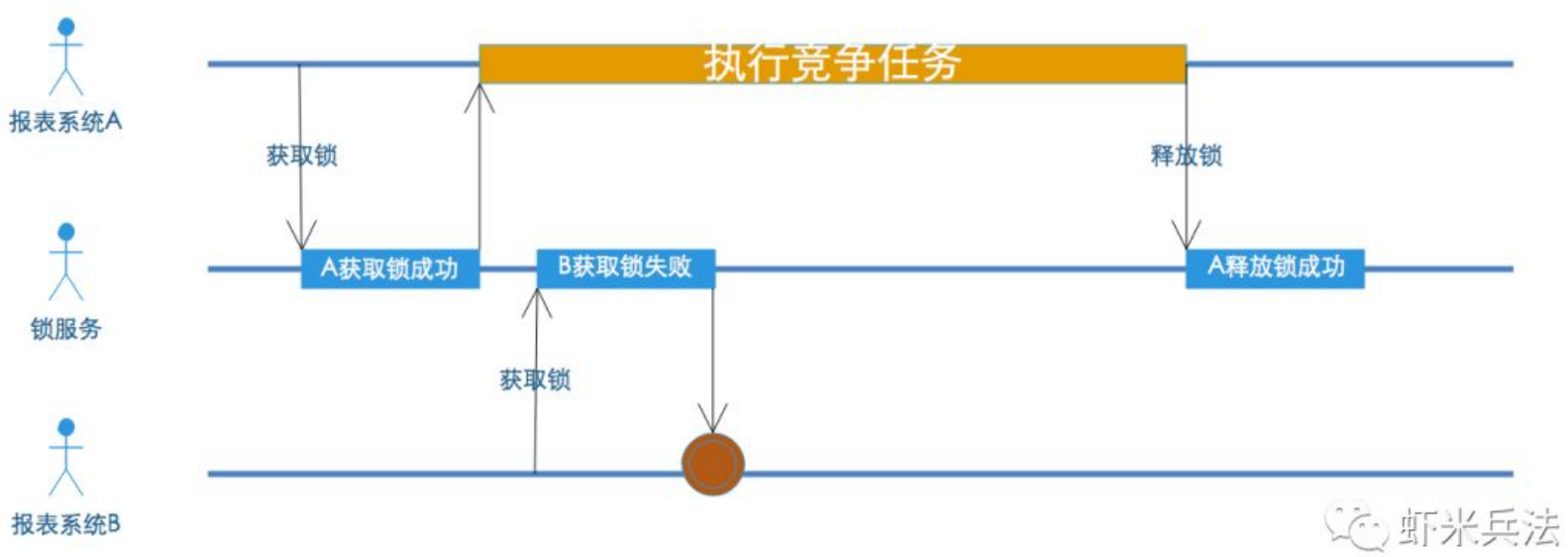
摘要：分布式架构设计如今在企业中被大量的应用，而在不同的分布式节点进行协同工作的时候，节点服务的时序、结果的正确性以及执行成本也成为了必须考虑的重要因素。其中竞态条件会导致执行结果的不正确，不同服务节点同时处理同一任务也将耗费不必需的系统资源，如果解决呢？方式之一可以选择分布式锁，本文介绍如果通过redis实现分布式锁，也欢迎大家和我一起讨论。

分布式锁的基本应用场景和设计原则

我们先来看一个简单的案例：有三个服务，一个是订单服务orderService，一个是报表服务（reportService），一个是推送服务（pushService），每个服务都横向部署在2个节点上。报表服务每天凌晨12点需要从订单服务拉取订单数据并生成报表，并且在每天早上8点通过推送服务向用户发送新生成的数据报表，需要如何设计这个流程？

首先我们需要了解该流程的两个关键点，第一，报表服务的2个节点只能有一个节点生成报表，否则会浪费系统资源，该关键点没有高可靠的要求（重复覆盖生成并不会得到错误结果）；第二，向同一个用户推送该数据报表也只能有一个节点去执行，否则用户会收到两份一样的报表，该关键点有高可靠要求。

我们可以从两个关键点中提取一个相同点，必须要设置一把锁，获的该锁的节点才能执行指定的任务。同时还能提取到一个不同点，那就是两种场景对获取锁的依赖程度不一致。我们来对该流程进行简单建模：



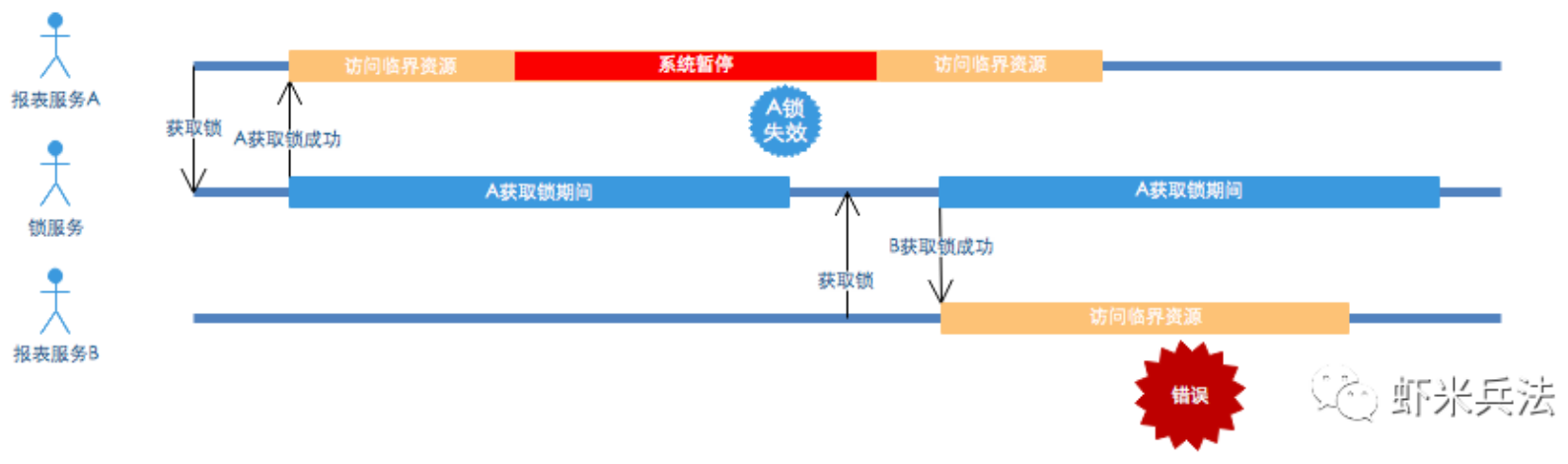
通过上图的流程已经可以实现简单可靠的锁机制，当然这是有前提的。

首先锁服务必须足够稳定，假设无法获取锁，那么竞争任务的将无法执行。其次，执行竞争任务的过程不能够死锁或者无限等待，否则将无法释放锁且改任务也无法执行完成。所以在设计锁的时候还需要考虑两个因素：锁必须要有过期时间及获取及释放锁过程的高可用或者锁错误时的异常处理。

所以，归纳一下分布式锁在设计时通常要考虑的几个要素是:

1. 分布式锁一定要保证多客户端竞争临界资源时的绝对互斥;
2. 分布式锁要设计一定的超时时间，防止在获得锁的服务阻塞或者崩溃引起的锁无法释放;
3. 分布式要针对业务场景设计锁机制异常降级措施，防止因为锁获取错误导致无法获取临界资源的后果。

关于第2点的要素，还有一些要注意的东西，假设报表服务A在获取到锁之后，出现了很长的FULL GC，系统出现暂停，在此期间，锁已经超时了，报表服务B又重新拿到了锁并向用户发送了报表，在客户端AFull GC结束后，同样再去执行报表发送任务，就会导致执行结果出错。



这种场景往往需要个性化的处理，现在业界大部分的分布式锁都会出现这种情况，因为系统暂停导致的锁失效往往很难去避免，因为系统暂停可能出现在任何时候。 通常情况下，我们需要预估访问竞争资源的时间，确定好超时时间并在访问结束后进行数据比对和必要的数据补偿。

Redis具体实现分布式锁

在redis命令集合中，有一个命令叫做SETNX，具体命令格式是:

```
SETNX key value
```

该命令的作用是如果key存在，则什么都不做，并且返回0，如果key不存在则将key的值设置成value，并且返回1，该命令是原子性的。我们可以利用该命令来实现分布式锁。

1. 获取锁：获取当前的timestamp，并将客户端ID作为key，该timestamp作为value调用SETNX，并设置锁的TTL，处理获取锁的异常。
2. 确认锁状态，如果成功获取锁，则访问临界资源，否则根据业务场景间隔一定时间再次尝试获取锁。
3. 访问临界资源
4. 释放锁

```
//获取锁

timeStamp = getCurrentTimeStamp();

try{

    lock=SET CLIENT_ID timeStamp NX PX TIMEOUT;

}catch(Exception e){

    //处理获取锁的异常

    return;

}

try{

    if(lock == 0){

        return;

    }else{

        //访问临界资源

        do();

    }

}finally{
```

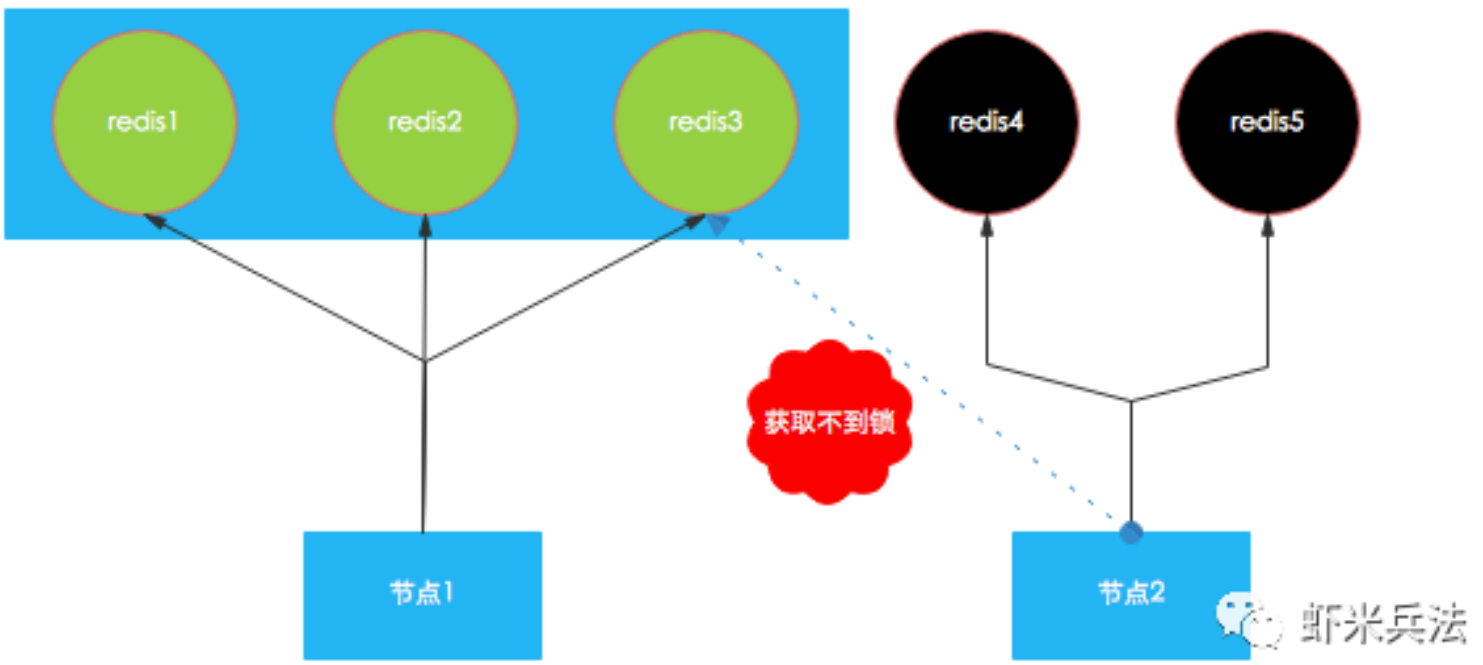
```
    //释放锁

    del CLIENT_ID;

}
```

这种实现分布式锁的方式是很多开发者最喜欢用的，但是如何保证redis的可用性呢，如果我们使用一个redis节点，当其因为不可控原因宕机时，锁机制将不可用。有人可能会说，可以使用redis主从集群复制，主挂了，从可以接替上，但是这估计依然不能解决问题，因为redis主从复制是异步的，谁能保证主挂了，从节点上一定有锁数据呢?

redis官网上介绍了一种**red lock**算法，该算法弃用了单redis节点，采用N个（官网推荐5个）独立的redis节点作为锁服务，客户端要获取锁，必须向N/2+1（绝大部分）节点成功申请锁后，才能访问临界资源。



但是该算法中获取锁的过程变的复杂了，时间也就越不可控，假设从redis1节点获取锁成功开始到从redis（N/2+1）获取锁成功结束到时间为SPACETIME，锁到有效时间不再是key到TTL，而是:

```
REMAIN_TIME=TTL-SPACETIME
```

当SPACETIME比较大时，客户端非常有可能获取到一个已经失效到锁，所以在获取锁之后red lock算法需要再次验证锁是否失效。

```
//获取锁
```

```
timeStamp = getCurrentTimeStamp();
//向N/2+1个节点申请锁
int successLockNum=0;
boolean lockSuccess=false;
for(int i=1;i<5;i++){
    try{
        lock=SET CLIENT_ID timeStamp NX PX TIMEOUT;
        if(lock == 1 && ++successLockNum == N/2+1){
            lockSuccess = true;
            break;
        }
    }catch(Exception e){
        //处理获取锁的异常
        return;
    }
}
//验证获取锁是否成功
if(!successLockNum){
    //获取锁失败
    return;
}
//验证获取到到锁是否是无效锁
nowTimeStamp = getCurrentTimeStamp();
if(nowTimeStamp-timeStamp>TTL){
    //无效锁
    return;
}
try{
    //访问临界资源
```

```
    do();
}finally{
    //释放锁
    del CLIENT_ID;
}
```

后续

用Redis来实现分布式锁机制在业界非常常用，但是我们在应用过程中一定要注意实现锁到超时避免死锁以及因为服务暂停导致锁失效到情况，每种情况到解决方案需要个性化到去解决。Red lock算法在一定程度上解决了分布式锁服务到可英雄问题，但是打来了系统复杂度，同时也有人在质疑了该算法，有兴趣到可以在搜索引擎搜索。本文就到这里，如有错误，欢迎指正。