**一、为什么需要分布式锁：**

本地锁保证本地的多个线程不会产生线程不安全。

在分布式环境下，也有类似的需求，例如选主，有且只有一个节点执行任务。多个进程可能抢一个分布式锁。

**二、分布式锁的目的：**

1. 效率。一个任务不会在对等节点上执行多次
2. 正确性。保证一个任务有且只有执行一次，避免相互覆盖

**三、分布式锁的特点：**

分布式锁也具备跟本地锁类似的特点。但分布式锁分为两个维度，一个是进程（不同的节点），一个是线程，同一个进程中不同线程。

互斥性：不同应用不同线程之间应是互斥的。

可重入性：同一个节点的同一个线程可以多次获得锁。获得多次也要释放多次。

锁超时：获取锁时有超时时间。

高可用：当分布式锁的一个或多个节点挂掉后，仍然能保证锁的状态，也能提供正常的锁服务。

高效：加锁和解锁耗时短。

阻塞和非阻塞：阻塞获取锁即lock()，非阻塞获取锁tryLock()。

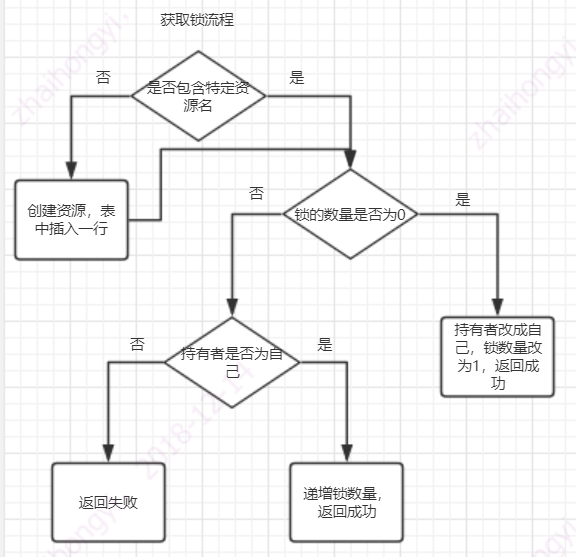
公平和非公平：公平指的是按获取锁的顺序获取锁，非公平是当锁释放后，不一定谁会拿到锁，可能是后面请求的线程。

**四、常见的分布式锁：**

4.1 MySQL：

表可以设计为id, resourceName, holder, count。

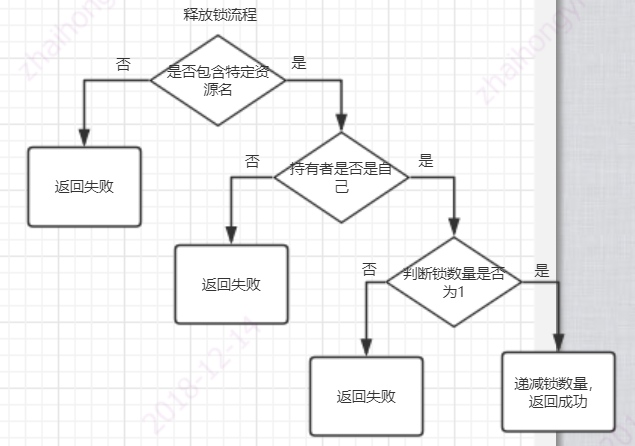
获取锁的流程：



这一系列操作包含了select和update，可以使用select for update行锁。同时整个操作应使用事务操作，保证原子性。也可以人为使用互斥锁保证资源独占。使用互斥锁就无需使用select for update。

对于阻塞和非阻塞的实现，阻塞可以用死循环一直等到获取锁，否则等待一段时间，继续尝试获取。非阻塞则尝试获取一次，获取不到返回false。同时非阻塞还可以设置超时时间，在超时之前会一直循环获取。

释放锁流程：



如果一个应用节点获取锁执行任务，在释放锁之前挂掉，那后续请求锁都会阻塞或拒绝，这不是使用数据库造成的问题，只能由其他调度任务根据获取锁的时长强行释放锁。

优点：无需维护其他组件

缺点：性能较差，自己维护获取、释放、超时逻辑。

可以使用乐观锁，在并发较低的情况下避免行锁。即在字段中增加version字段，每次更新操作都会递增version，只有在select时的version跟update时version相同，才更新。

4.2 ZK

ZK的临时Sequence节点帮助实现分布式锁，因为其子节点的Sequence是互斥的，即一个线程只能创建一个子节点，节点名称为xxx\_0001，另一个线程只能创建xxx\_0002节点，sequence依次递增。我们可以使用ZK的客户端实现分布式锁的获取和释放。

官方客户端Curator提供了分布式锁的实现。InterProcessMutex是Curator实现的可重入锁，我们可以通过下面的一段代码实现我们的可重入锁:

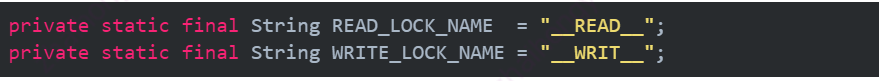


基于ZK的重入锁，获取锁的数量存在客户端本地，维护一个ConcurrentHashMap，每次获取锁都会插入或更新记录，记录的KV分别是资源名和获取锁的数量。

其获取锁的流程：在指定路径下创建临时有序节点，再获取当前路径下所有子节点，如果创建的节点序列号最小，则获取锁。其他没获取锁的会注册监听事件，监听子节点数量的变化。如果释放锁，则第一个节点被删除，ZK会下发通知，因为有序，所以只有一个序号+1的子节点对应的线程会唤醒。ZK默认就是公平锁。

释放流程：如果可重入，则在本地先递减获取锁次数，如果减到0，则删除节点并删除本地KV。

以上描述的都是互斥锁，只有一个线程能拿到锁。Curator还提供了读写锁。InterProcessReadWriteLock。在创建临时有序节点时，节点名称都会增加是读操作还是写操作前缀。



读锁会监听最近的一个写锁的状态，如果删除，多个读锁都会释放。但写锁跟读和其他写锁都是互斥的，所以它只监听并判断自己是否是序列最小的节点。

使用ZK无需配置锁超时，因为如果节点挂掉，zk客户端和服务端的session超时后，服务端自动删除临时节点。

使用Curator不仅能实现分布式锁，还能实现选主，选主还包括一次选主和主备切换（主挂掉，临时节点消失，下一个备机称为主机）。他们的逻辑是相同的，只不过暴露的接口不同。

优点：无需实现复杂逻辑，读写锁，公平锁

缺点：性能较差

4.3 Redis