Redis分布式锁的正确实现方式：

单节点redis：1、只能由单个客户端持有锁 2、只能有拥有锁的客户端解锁 3、不会死锁（锁有超时时间）

加锁解锁时，需要保证操作的原子性，包括设置过期时间。

jedis.set(lockKey, requestId, "NX", "PX", expireTime);

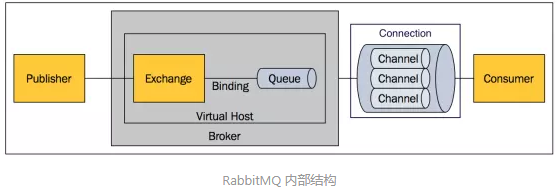
String script = "if redis.call('get', KEYS[1]) == ARGV[1] then return redis.call('del', KEYS[1]) else return 0 end";

Object result = jedis.eval(script, Collections.singletonList(lockKey), Collections.singletonList(requestId));

多节点：redission组件。

Redis是单线程的。

数据结构： String、Hash、List、Set、有序集合SortedSet。



Publisher -> Exchange -> Binding -> Queue -> Channel -> Consumer

Exchange 类型：

1、direct： routing key和Binding中的 binding key 完全一致。

2、fanout：绑定所有队列。

3、topic：模式匹配的方式。

routing\_key 和 binding\_key 的最大长度为255字节。

作用：

1、解耦：降低系统之间的耦合性。

2、异步：将非实时的业务逻辑异步执行。

3、削峰：并发量大时，转移部分请求到消息队列，延迟处理。

缺点：

1、降低系统可用性。

2、提升系统复杂性。

生产者重复发布/丢失消息：

Rabbitmq使用了事务和confirm机制来确保正确。发布消息后，收到确认消息才可继续发送，失败则重试。使用 confirm 机制来优化生产消息的确认（可以持续发布消息，但会批量回复确认）。

消息持久化机制：

与confirm机制配合使用，在消息持久化磁盘后才返回确认消息。

消息重复消费：

重复消费条件苛刻，在大多数场景下不会触发该条件。一般在任务超时，或者没有及时返回状态，引起任务重新入队列，重新消费。

解决方案：消费者维护一个map记录任务的执行情况（需要根据业务的不同采取适当的解决方案）。

消费者消息丢失：

Rabbimq默认为自动ack，修改为消费者消费完成，手动ack。消费成功后返回确认消息。

死信队列：

利用 dlx，当消息在一个队列中变成死信 (dead message) 之后，它能被重新 publish 到另一个 exchange，这个 exchange 就是 dlx。

消息变成死信的原因有：

1.消息被拒绝 (basic.reject / basic.nack) 并且 reQueue=false

2.消息 TTL 过期

3.队列达到最大长度了