分布式锁常用的有zookeeper和redis，项目中已经在使用redis了，所以就选择redis实现分布式锁。

分布式锁的特点：

1. 互斥性，保证任意时刻，只有一个客户端持有锁
2. 锁超时，即使一个客户端因为种种原因没有主动释放锁，也要保证其它客户端能加锁成功
3. 加锁和解锁必须是同一客户端，客户端不能把别人加的锁给释放了。

参考：<https://zhuanlan.zhihu.com/p/112016634>

目前redis版本已到达6.0了，所以不考虑redis 2.6.12之前的情形了

代码：lintcode/twitter/redis\_test.ipynb

加锁：

def acquire\_lock(conn, lock\_name, acquire\_timeout=3, lock\_timeout=2):

"""

基于 Redis 实现的分布式锁

:param conn: Redis 连接

:param lock\_name: 锁的名称

:param acquire\_timeout: 获取锁的超时时间，默认 3 秒

:param lock\_timeout: 锁的超时时间，默认 2 秒

"""

# 这个采用uuid生成的标识符，主要是在release\_lock时，不要删除掉不是由自己创建的锁

identifier = str(uuid.uuid4())

end = time.time() + acquire\_timeout

lock\_name = "string:lock:{}".format(lock\_name)

while time.time() < end:

"""

获取锁如果没有超时，其它客户端释放锁时，会删除掉名称为lock\_name的key，客户端就能重新设置

一个名称为lock\_name的key，另外设置了超时时间，防止死锁

"""

if conn.set(lock\_name, identifier, ex=lock\_timeout, nx=True):

return identifier

time.sleep(0.001)

return ""

释放锁：

def release\_lock(conn, lock\_name, identifier):

"""

释放锁

:param conn: Redis 连接

:param lockname: 锁的名称

:param identifier: 锁的标识

:return:

"""

unlock\_script = """

if redis.call("get",KEYS[1]) == ARGV[1] then

return redis.call("del",KEYS[1])

else

return 0

end

"""

lock\_name = "string:lock:{}".format(lock\_name)

# 调用redis内嵌的lua脚本，保证get和delete的原子操作

unlock = conn.register\_script(unlock\_script)

result = unlock(keys=[lock\_name], args=[identifier])

if result:

return True

else:

return False

超时时间：

上述锁的第一个比较麻烦的问题就是：超时时间的设置，如果获取锁的客户端因为某些耗时操作，导致超时，锁就会丢掉。一种解决办法就是尽可能的给更长的冗余时间，但这种方法不太完美。

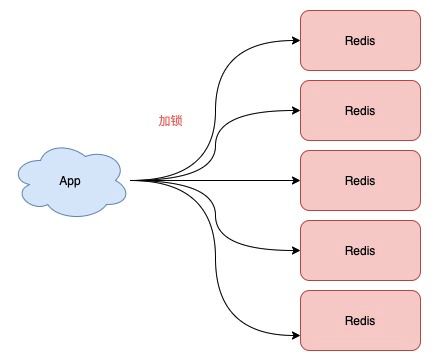
在Java中实现了Redisson，这是基于java的Redis SDK，它在加锁时启动了一个守护线程，定时去检测锁的失效时间，如果锁快要过期时，操作共享资源还未完成，就自动进行续期，重新设置过期时间。

第二个比较麻烦的问题：这种加锁方式只能在单个Redis服务器(可以在多个Redis服务器上选一个专门的用于分布式锁)或主从模式下，分布式主从模式时可能出现的锁丢失情况，单个Redis加锁服务器更有可能会宕机导致锁丢失。注意：这种加锁方式不能采用一致性hash算法，因为一致性hash算法仍然会有一小部分key失效，如果正好是加锁用的key，就会出现锁丢失的问题。

主从模式下，客户端在主库上加锁成功，这时主库宕机，这时主从未进行同步(主从复制是异步的)，发生了主从切换，从库升级为主库时，之前加的锁丢失。

Redis的作者提出了一个RedLock，核心思想是：

不再部署从库，只部署主库，而且至少部署5个以上的主库，并且主库之间相互独立。



整个流程分5步：

1. 客户端获取当前时间戳T1
2. 客户端依次向N个Redis发起SET请求(之前单个Redis加锁用到的SET命令)，且每个请求都会设置超时时间(毫秒级，远小于锁的有效时间)，如果某个Redis加锁失败(包括网络超时、锁被其它客户端持有等情况)，就立即向下一个Redis请求加锁
3. 如果客户端从超过半数的Redis加锁成功，即大于等于N/2+1，则再次获取当前时间戳T2，如图T2-T1<锁的过期时间，认为客户端加锁成功，否则认为加锁失败。
4. 加锁成功，执行某种操作，向所有Redis结点申请释放锁(lua脚本释放锁)
5. 加锁失败，向所有Redis结点申请释放锁(lua脚本释放锁)

加锁失败后的重试：应该在随机时间后重试，并且最好同一时刻把set命令并发给所有的Redis。

RedLock为什么要这么做？

1. 为什么要在多个实例上加锁？

本质上是为了容错，如果部分结点宕机了，剩下的结点加锁成功，仍然可以使用锁服务。

1. 为什么超过半数结点加锁成功，才算成功？

这样保证同一时刻只有1个客户端持有锁

1. 加锁成功后，为什么要判断超时？

因为加锁要操作多个结点，由于网络情况的复杂性，即使大多数结点加锁成功，但加锁累计消耗的时间超过了锁的有效时间，则可能会出现部分结点上的锁失效的问题，这个锁就没有意义了。

1. 为什么释放锁，要操作所有结点？

在某个Redis结点加锁失败，可能实际加锁成功，但由于网络原因没有读取到响应结果，导致判断它加锁失败。所以释放锁时，要操作所有结点，保证没有残留的锁(根据之前单例Redis释放锁的操作，不会出现把别的客户端加的锁释放掉的问题)。

RedLock引发的争论：

效率：如果web系统更看重效率，单例Redis锁就足够了，偶尔出现锁失效的问题，也不影响，例如邮件发送了两次。

正确性：RedLock多个结点的时钟是否同步对加锁有较大的影响，需要基础设施和运维来保证。

RedLock其实无法解决成功拿到锁之后，操作共享资源时发生了NPC()问题后，锁自动超时失效后，被别的客户端拿走的问题。

N：Network Delay，网络延迟

P：Process Pause，进程中止

C：Clock Drift，时钟漂移