Redis Distributed Lock

一、基本要求

1. 互斥，不论何时，只能一个客户端持有锁
2. 无死锁，即使一个持有锁的客户端崩溃，该锁最后也能释放
3. 容错，只要大部分redis节点是活跃的，分布式锁就能够正常申请和释放

二、单节点分布式锁

在实现单节点分布式锁的时候遇到一些问题，下文按照问题的顺序，总结出了以下几个版本和对应的缺陷说明，版本按照先后顺序，逐步迭代出相对较完善的单节点分布式锁版本。

2.1 版本一

相关命令

```
SETNX key value
设置key的value，如果存在此key则不进行任何操作，返回0，否则返回1

DEL key [key ...]
删除key，返回删除的key个数
```

锁流程

- 申请锁

  SETNX lock_key value

  返回1，设置成功 => 获取锁成功，进入临界区

  返回0，设置失败 => 申请锁失败，自旋或阻塞之

- 释放锁

  DEL lock_key

  返回1 => 释放锁成功

  返回0 => 释放锁失败

缺陷

考虑以下场景：

线程A申请到锁，在持有锁期间，线程A异常退出，lock_key没有删除，这会导致阻塞在此锁上及后来申请此锁的线程全部被永久阻塞。

2.2 版本二

为了保证要求2无死锁，考虑给lock加一个过去时间使其最终能够被释放

相关命令

```
SET key value EX seconds NX
设置key，同时加上一个过期时间seconds秒，如果存在此key则不进行任何操作，返回0，否则返回1

DEL key [key ...]
删除key，返回删除的key个数
```

锁流程

- 申请锁

  SET lock_key value EX seconds NX

  返回1，设置成功 => 获取锁成功，进入临界区

  返回0，设置失败 => 申请锁失败，自旋或阻塞之

- 释放锁

  1. DEL lock_key

     返回1 => 释放锁成功

     返回0 => 释放锁失败

  2. key过期，这里解决了版本一的缺陷，即使持有锁的线程没有正常释放锁，此锁依旧会在一定时间后过期释放，避免其它竞争线程永久等待

缺陷

1. 业务处理结束之前锁就过期了，此时多个线程会进入同一临界区。
2. 考虑以下场景：

   step 1. Client A 获得lock，并设置过期时间1000秒

   step 2. A遇到一个阻塞操作直到2000秒时才被唤醒继续执行

   step 3. 此时Client B申请lock并成功（因为此lock已经expire）

   step 4. A执行到释放锁的代码，问题出现，A把B的锁释放了

2.3 版本三

解决版本二缺陷2，为了避免释放不属于自身线程的锁，考虑给锁加一个唯一标识，表示持有锁的线程，可以在lock_key的value字段设置。

锁流程

- 申请锁

  SET lock_key unique_code EX seconds NX

  返回1，设置成功 => 获取锁成功，进入临界区

  返回0，设置失败 => 申请锁失败，自旋或阻塞之

- 释放锁

  ```
  unique_code = value(lock_key)
  if cur_thread_code == unique_code:
      del(lock_key)
  ```

  释放锁的过程变为多个操作，为了保证unlock的原子性，考虑使用lua脚本执行：

```
if redis.call('GET', KEYS[1]) == ARGV[1]
    then return redis.call('DEL', KEYS[1])
    else return 0
end
```

使用eval命令：`EVAL script 1 key, LOCK_KEY, UNIQUE_CODE`

为什么unlock需要原子性？

考虑以下情形

1. A获取锁成功。
2. A访问共享资源。
3. A释放锁，先GET随机字符串的值并且与自己的值相等
4. A由于某个原因阻塞住了很长时间，或者网络出现很大延迟
5. 过期时间到了，锁自动释放了。
6. B获取到了对应同一个资源的锁。
7. A发送的`DEL`到达了服务器，释放掉了B持有的锁。

- Unique Code

  Tedis采用的是Unix时间戳+host name+host address方式来实现

2.4 版本四

解决版本二缺陷1，为了避免业务逻辑执行完之前锁就过期释放了，考虑在业务执行过程中开一个单独线程延长锁的过期时间，业务逻辑执行完之后停止此线程。

```
lock() {
    ...
    // 加锁成功时开启更新线程
    updateThread.start();
    ...
}

unlock() {
    // 释放锁时关闭更新线程
    // 此处通过把Runnable任务的循环标志设置为false以退出循环结束任务
    task.stop();
}
```

其实，解决版本的缺陷1，缺陷2也不复存在了。

2.5 总结

单节点的分布式锁给出了利用redis实现资源互斥的方案，但是它要求一个理想的redis实例，也就是说实现但节点分布式锁的这个redis实例需要始终保持可用，完全安全，所以为了能够在实际生产环境中使用redis的分布式锁，就考虑到使用redis集群来保证系统的容错性，保证高可用。