基于 Redis 的分布式锁

前言

分布式锁在分布式应用中应用广泛，想要搞懂一个新事物首先得了解它的由来，这样才能更加的理解甚至可以举一反三。

首先谈到分布式锁自然也就联想到分布式应用。

在我们将应用拆分为分布式应用之前的单机系统中，对一些并发场景读取公共资源时如扣库存，卖车票之类的需求可以简单的使用同步或者是加锁就可以实现。

但是应用分布式了之后系统由以前的单进程多线程的程序变为了多进程多线程，这时使用以上的解决方案明显就不够了。

因此业界常用的解决方案通常是借助于一个第三方组件并利用它自身的排他性来达到多进程的互斥。如：

- 基于 DB 的唯一索引。
- 基于 ZK 的临时有序节点。
- 基于 Redis 的 `NX EX` 参数。

这里主要基于 Redis 进行讨论。

实现

既然是选用了 Redis，那么它就得具有排他性才行。同时它最好也有锁的一些基本特性：

- 高性能(加、解锁时高性能)
- 可以使用阻塞锁与非阻塞锁。
- 不能出现死锁。
- 可用性(不能出现节点 down 掉后加锁失败)。

这里利用 `Redis set key` 时的一个 NX 参数可以保证在这个 key 不存在的情况下写入成功。并且再加上 EX 参数可以让该 key 在超时之后自动删除。

所以利用以上两个特性可以保证在同一时刻只会有一个进程获得锁，并且不会出现死锁(最坏的情况就是超时自动删除 key)。

加锁

实现代码如下：

```
    private static final String SET_IF_NOT_EXIST = "NX";
    private static final String SET_WITH_EXPIRE_TIME = "PX";

    public  boolean tryLock(String key, String request) {
        String result = this.jedis.set(LOCK_PREFIX + key, request, SET_IF_NOT_EXIST, SET_WITH_EXPIRE_TIME, 10 * TIME);

        if (LOCK_MSG.equals(result)){
            return true ;
        }else {
            return false ;
        }
    }
```

注意这里使用的 jedis 的

```
String set(String key, String value, String nxxx, String expx, long time);
```

api。

该命令可以保证 NX EX 的原子性。

一定不要把两个命令(NX EX)分开执行，如果在 NX 之后程序出现问题就有可能产生死锁。

阻塞锁

同时也可以实现一个阻塞锁：

```
    //一直阻塞
    public void lock(String key, String request) throws InterruptedException {

        for (;;){
            String result = this.jedis.set(LOCK_PREFIX + key, request, SET_IF_NOT_EXIST, SET_WITH_EXPIRE_TIME, 10 * TIME);
            if (LOCK_MSG.equals(result)){
                break ;
            }

              //防止一直消耗 CPU
            Thread.sleep(DEFAULT_SLEEP_TIME) ;
        }

    }

     //自定义阻塞时间
     public boolean lock(String key, String request,int blockTime) throws InterruptedException {

        while (blockTime >= 0){

            String result = this.jedis.set(LOCK_PREFIX + key, request, SET_IF_NOT_EXIST, SET_WITH_EXPIRE_TIME, 10 * TIME);
            if (LOCK_MSG.equals(result)){
                return true ;
            }
            blockTime -= DEFAULT_SLEEP_TIME ;

            Thread.sleep(DEFAULT_SLEEP_TIME) ;
        }
        return false ;
    }
```

解锁

解锁也很简单，其实就是把这个 key 删掉就万事大吉了，比如使用 `del key` 命令。

但现实往往没有那么 easy。

如果进程 A 获取了锁设置了超时时间，但是由于执行周期较长导致到了超时时间之后锁就自动释放了。这时进程 B 获取了该锁执行很快就释放锁。这样就会出现进程 B 将进程 A 的锁释放了。

所以最好的方式是在每次解锁时都需要判断锁**是否是自己**的。

这时就需要结合加锁机制一起实现了。

加锁时需要传递一个参数，将该参数作为这个 key 的 value，这样每次解锁时判断 value 是否相等即可。

所以解锁代码就不能是简单的 `del` 了。

```
    public  boolean unlock(String key,String request){
        //lua script
        String script = "if redis.call('get', KEYS[1]) == ARGV[1] then return redis.call('del', KEYS[1]) else return 0 end";

        Object result = null ;
        if (jedis instanceof Jedis){
            result = ((Jedis)this.jedis).eval(script, Collections.singletonList(LOCK_PREFIX + key), Collections.singletonList(request));
        }else if (jedis instanceof JedisCluster){
            result = ((JedisCluster)this.jedis).eval(script, Collections.singletonList(LOCK_PREFIX + key), Collections.singletonList(request));
        }else {
            //throw new RuntimeException("instance is error") ;
            return false ;
        }

        if (UNLOCK_MSG.equals(result)){
            return true ;
        }else {
            return false ;
        }
    }
```

这里使用了一个 `lua` 脚本来判断 value 是否相等，相等才执行 del 命令。

使用 `lua` 也可以保证这里两个操作的原子性。

因此上文提到的四个基本特性也能满足了：

- 使用 Redis 可以保证性能。
- 阻塞锁与非阻塞锁见上文。
- 利用超时机制解决了死锁。
- Redis 支持集群部署提高了可用性。

使用

我自己有撸了一个完整的实现，并且已经用于了生产，有兴趣的朋友可以开箱使用:

maven 依赖：

```
<dependency>
    <groupId>top.crossoverjie.opensource</groupId>
    <artifactId>distributed-redis-lock</artifactId>
    <version>1.0.0</version>
</dependency>
```

配置 bean：

```
@Configuration
public class RedisLockConfig {

    @Bean
    public RedisLock build(){
        RedisLock redisLock = new RedisLock() ;
        HostAndPort hostAndPort = new HostAndPort("127.0.0.1",7000) ;
        JedisCluster jedisCluster = new JedisCluster(hostAndPort) ;
        // Jedis 或 JedisCluster 都可以
        redisLock.setJedisCluster(jedisCluster) ;
        return redisLock ;
    }

}
```

使用：

```
    @Autowired
    private RedisLock redisLock ;
```

```
    public void use() {
        String key = "key";
        String request = UUID.randomUUID().toString();
        try {
            boolean locktest = redisLock.tryLock(key, request);
            if (!locktest) {
                System.out.println("locked error");
                return;
            }


            //do something

        } finally {
            redisLock.unlock(key,request) ;
        }

    }
```

使用很简单。这里主要是想利用 Spring 来帮我们管理 RedisLock 这个单例的 bean，所以在释放锁的时候需要手动(因为整个上下文只有一个 RedisLock 实例)的传入 key 以及 request(api 看起来不是特别优雅)。

也可以在每次使用锁的时候 new 一个 RedisLock 传入 key 以及 request，这样倒是在解锁时很方便。但是需要自行管理 RedisLock 的实例。各有优劣吧。

项目源码在：

https://github.com/crossoverJie/distributed-lock-redis

欢迎讨论。

单测

在做这个项目的时候让我不得不想提一下**单测**。

因为这个应用是强依赖于第三方组件的(Redis)，但是在单测中我们需要排除掉这种依赖。比如其他伙伴 fork 了该项目想在本地跑一遍单测，结果运行不起来：

1. 有可能是 Redis 的 ip、端口和单测里的不一致。
2. Redis 自身可能也有问题。
3. 也有可能是该同学的环境中并没有 Redis。

所以最好是要把这些外部不稳定的因素排除掉，单测只测我们写好的代码。

于是就可以引入单测利器 `Mock` 了。

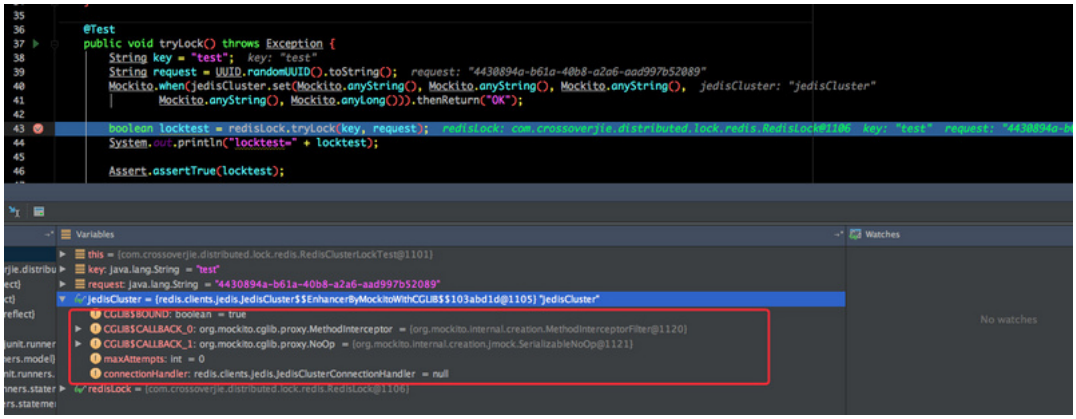
它的想法很简答，就是要把你所依赖的外部资源统统屏蔽掉。如：数据库、外部接口、外部文件等等。

使用方式也挺简单，可以参考该项目的单测：

```
    @Test
    public void tryLock() throws Exception {
        String key = "test";
        String request = UUID.randomUUID().toString();
        Mockito.when(jedisCluster.set(Mockito.anyString(), Mockito.anyString(), Mockito.anyString(),
                Mockito.anyString(), Mockito.anyLong())).thenReturn("OK");

        boolean locktest = redisLock.tryLock(key, request);
        System.out.println("locktest=" + locktest);

        Assert.assertTrue(locktest);

        //check
        Mockito.verify(jedisCluster).set(Mockito.anyString(), Mockito.anyString(), Mockito.anyString(),
                Mockito.anyString(), Mockito.anyLong());
    }
```

这里只是简单演示下，可以的话下次仔细分析分析。

它的原理其实也挺简单，debug 的话可以很直接的看出来：



这里我们所依赖的 JedisCluster 其实是一个 `cglib 代理对象`。所以也不难想到它是如何工作的。

比如这里我们需要用到 JedisCluster 的 set 函数并需要它的返回值。

Mock 就将该对象代理了，并在实际执行 set 方法后给你返回了一个你自定义的值。

这样我们就可以随心所欲的测试了，**完全把外部依赖所屏蔽了**。

总结

至此一个基于 Redis 的分布式锁完成，但是依然有些问题。

- 如在 key 超时之后业务并没有执行完毕但却自动释放锁了，这样就会导致并发问题。
- 就算 Redis 是集群部署的，如果每个节点都只是 master 没有 slave，那么 master 宕机时该节点上的所有 key 在那一时刻都相当于是释放锁了，这样也会出现并发问题。就算是有 slave 节点，但如果在数据同步到 salve 之前 master 宕机也是会出现上面的问题。

感兴趣的朋友还可以参考 Redisson 的实现。