Redission分布式锁

如果聊到了分布式系统这块的东西。通常面试官都会从服务框架（Spring Cloud、Dubbo）聊起，一路聊到分布式事务、分布式锁、ZooKeeper等知识。

如果在公司里落地生产环境用分布式锁的时候，一定是会用开源类库的，比如Redis分布式锁，一般就是用Redisson框架就好了，非常的简便易用。

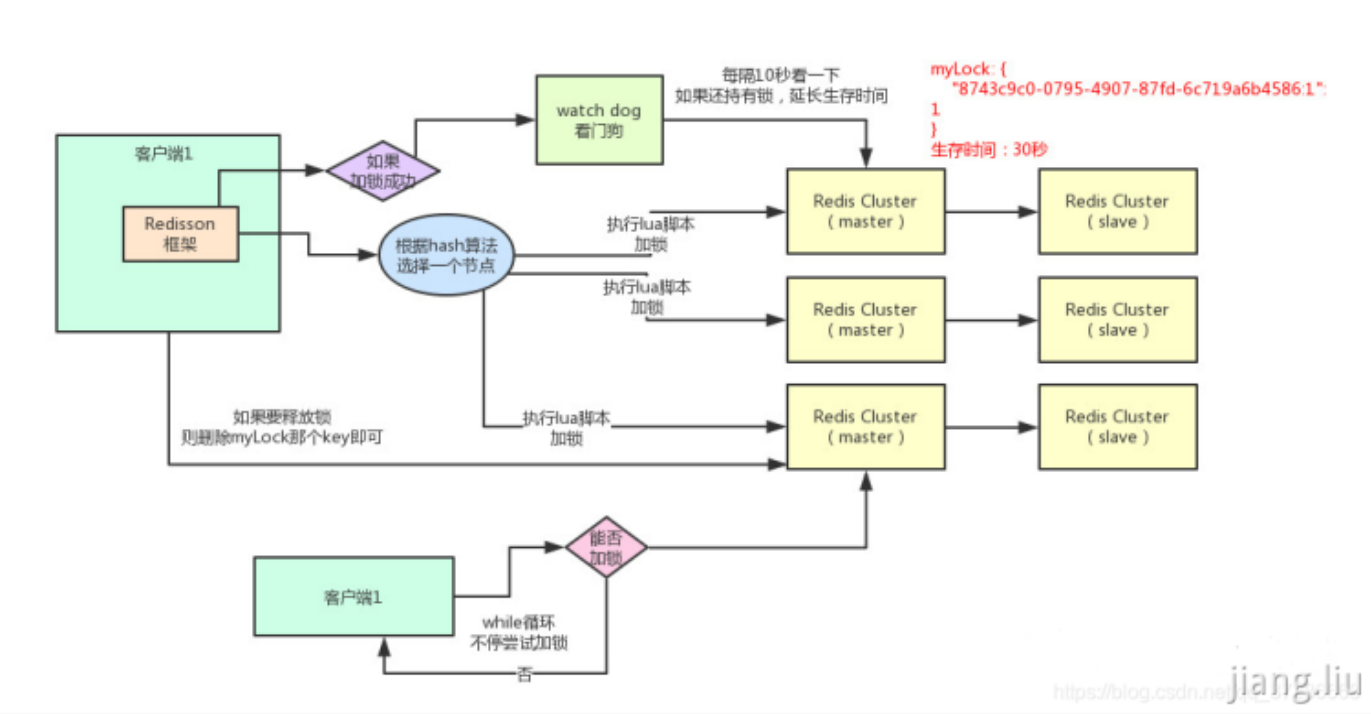
可以去看看Redisson的官网，看看如何在项目中引入Redisson的依赖，然后基于Redis实现分布式锁的加锁与释放锁。下面是一个简单得Redisson分布式锁的实现伪代码：

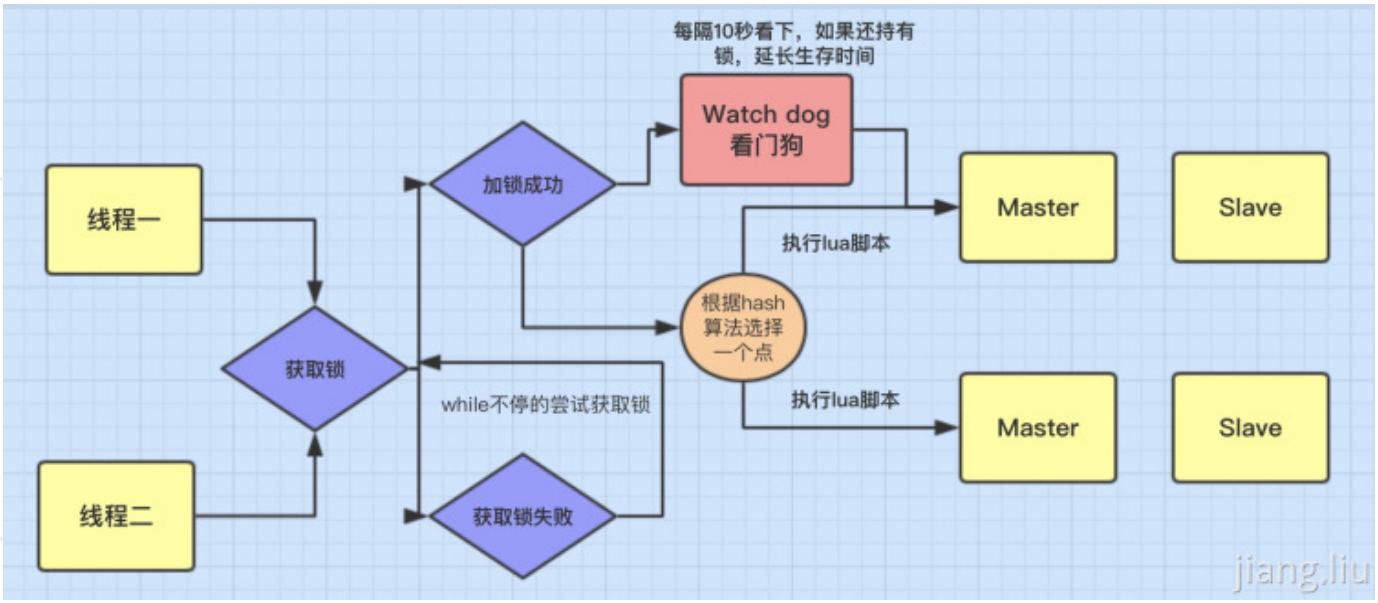
Java 复制代码

```
RLock lock = redisson.getLock("myLock"); //获取锁
lock.lock();  //上锁
...  // 业务代码
lock.unlock();  //释放锁
```

代码很简单，而且它还支持redis单实例、redis哨兵、redis cluster、redis master-slave等各种部署架构，都可以给你完美实现。

Redis分布式锁的底层原理





1）加锁机制

现在某个客户端需要加锁。如果该客户端面对的是一个redis cluster 集群，它就需要根据hash节点去选择一台机器。然后就会发送一个lua脚本。脚本代码如下图所示：

```
"if (redis.call('exists', KEYS[1]) == 0) then " +
    "redis.call('hset', KEYS[1], ARGV[2], 1); " +
    "redis.call('pexpire', KEYS[1], ARGV[1]); " +
    "return nil; " +
"end; " +
"if (redis.call('hexists', KEYS[1], ARGV[2]) == 1) then " +
    "redis.call('hincrby', KEYS[1], ARGV[2], 1); " +
    "redis.call('pexpire', KEYS[1], ARGV[1]); " +
    "return nil; " +
"end; " +
"return redis.call('pttl', KEYS[1]);"
```


为什么要用lua脚本?

因为lua脚本可以保证原子性!

上面的lua脚本代表的意思：

`KEYS[1]`代表的是你加锁的那个key，比如说：`RLock lock = redisson.getLock("myLock");`这里你自己设置了加锁的那个锁key就是“myLock”。

`ARGV[1]`代表的就是锁key的默认生存时间，默认30秒。

`ARGV[2]`代表的是加锁的客户端的ID，类似于这样：`8743c9c0-0795-4907-87fd-6c719a6b4586:1`

第一段if判断语句，就是用“exists myLock”命令判断一下，如果你要加锁的那个锁key不存在的话，你就进行加锁。

如何加锁呢？很简单，用下面的命令：

```
Java | 复制代码
hset  myLock
  8743c9c0-0795-4907-87fd-6c719a6b4586:1 1
```

通过这个命令设置一个hash数据结构，这行命令执行后，会出现一个类似下面的数据结构：

```
myLock:
{
    “8743c9c0-0795-4907-87fd-6c719a6b4586:1”: 1
}
```


上述就代表 `8743c9c0-0795-4907-87fd-6c719a6b4586:1` 这个客户端对“myLock”这个锁key完成了加锁。接着会执行 `pexpire myLock 30000` 命令，设置myLock这个锁key的生存时间是30秒。到此为止加锁就完成了。

2）锁互斥机制

如果客户端2来尝试加锁，执行了同样的一段lua脚本。第一个 if 判断会执行“exists myLock”，发现myLock这个锁key已经存在了。

接着第二个if判断，判断一下，myLock锁key的hash数据结构中，是否包含客户端2的ID，但是明显不是的，因为那里包含的是客户端 1 的 ID 。

所以，客户端 2 会获取到 `pttl myLock` 返回的一个数字，这个数字代表了 myLock 这个锁key的剩余生存时间。比如还剩15000毫秒的生存时间。此时客户端2会进入一个while循环，不停的尝试加锁。

3）watch dog自动延期机制

客户端 1 加锁的 key 默认生存时间才 30 秒，如果超过了 30 秒，客户端 1 还想一直持有这把锁，怎么办呢？

只要客户端 1 一旦加锁成功，就会启动一个 watch dog 看门狗，**他是一个后台线程，会每隔10秒检查一下**，如果客户端 1没有释放锁的操作，那么就会不断的延长锁 key 的生存时间（watch dog默认续期30s）。

> 正常这个看门狗线程是不启动的，还有就是这个看门狗启动后对整体性能也会有一定影响，所以不建议开启看门狗。

4）可重入加锁机制

那如果客户端1都已经持有了这把锁了，结果可重入的加锁会怎么样呢？代码如下图所示：

继续分析下lua脚本，第一个if判断肯定不成立，“exists myLock”会显示锁key已经存在了。第二个if判断会成立，因为myLock的hash数据结构中包含的那个ID，就是客户端1的那个ID，也就是“8743c9c0-0795-4907-87fd-6c719a6b4586:1”。

此时就会执行可重入加锁的逻辑，他会用：

```java
incrby myLock
  8743c9c0-0795-4907-87fd-6c71a6b4586:1 1
```

通过这个命令，对客户端1的加锁次数，累加1。

此时myLock数据结构变为下面这样：

这个myLock的hash数据结构中的这个客户端ID，就对应着它加锁的次数。

5）释放锁的机制

如果执行lock.unlock()，就可以释放分布式锁，此时的业务逻辑也是非常简单的。就是每次都对myLock数据结构中的那个加锁次数减1。

如果发现加锁次数是0了，说明这个客户端已经不再持有锁了，此时就会用“del myLock”命令，从redis里删除这个key。

然后呢，另外的客户端2就可以尝试完成加锁了。这就是所谓的分布式锁的开源Redisson框架的实现机制。

一般我们在生产系统中，可以用Redisson框架提供的这个类库来基于redis进行分布式锁的加锁与释放锁。

6）缺点

这种方案，一旦发生redis master宕机，主备切换，redis slave变为了redis master。接着就会导致，客户端2来尝试加锁的时候，在新的redis master上完成了加锁，而客户端1也以为自己成功加了锁。此时就会导致多个客户端对一个分布式锁完成了加锁。

这时系统在业务语义上一定会出现问题，导致各种脏数据的产生。

所以这个就是redis cluster，或者是redis master-slave架构的主从异步复制导致的redis分布式锁的最大缺陷：**在redis master实例宕机的时候，Redission可能导致多个客户端同时完成加锁**。