分布式锁

> ## 什么是锁?
>
> 使用锁的目的是为了控制程序的执行顺序，防止共享资源被多个线程同时访问。为了实现多个线程在一个时刻同一个代码块只能有一个线程可执行，那么需要在某个地方做个标记，这个标记必须每个线程都能看到，当标记不存在时可以设置该标记，其余后续线程发现已经有标记了则等待拥有标记的线程结束同步代码块取消标记后再去尝试设置标记。这个标记可以理解为锁。
>
> 不同地方实现锁的方式也不一样，只要能满足所有线程都能看得到标记即可。如 Java 中 synchronize 是在对象头设置标记，Lock 接口的实现类基本上都只是某一个 volitile 修饰的 int 型变量其保证每个线程都能拥有对该 int 的可见性和原子修改，linux 内核中也是利用互斥量或信号量等内存数据做标记。
>
> 单机应用中，如果我们想要控制某个时刻只有一个线程去执行某段代码或者修改某个变量，可以使用JDK自带的锁，比如Synchronized，ReentrantLock等。但是在微服务体系中，往往单个服务会部署在多个节点上，这个时候想要控制某段代码或者变量同一时刻只有一个线程可以访问的话，使用JDK自带的锁就不行了，这个时候就需要引入分布式锁。

场景：我们的程序需要在用户订单状态变更的时候给用户微信小程序发一条消息，当然，这个接口也是微信提供的，为了校验权限，微信提供的接口需要调用方传入access_token（access_token的有效期只有两个小时，并且一旦调用获取access_token的接口，上一次获取的access_token就失效了）。这就需要我们在调用获取access_token的接口的时候，保证同一时刻只有一个线程执行。否则，可能A线程刚刚获取到access_token，准备使用这个access_token去发送消息的时候，B线程突然也进来去执行获取access_token的操作，这就导致A线程获取的access_token失效了，发送消息也会失败。

分布式锁如何实现?

如上图：node1，node2，node3是部署在不同物理节点的同一服务，同一时刻，三个请求分别打到三个节点上，

每个节点都会由本结点的JVM启动一条线程去处理这个请求，假设三个线程thread1，thread2，thread3正好在同一时刻到达统一代码段，但是该段代码同一时刻只能有一个线程执行，这个时候就需要让这三个线程去竞争这个锁，只有获取到锁的线程才可以去执行这段代码，其他两个线程需要去阻塞等待或者直接返回。

很明显，这个分布式锁需要对这三个线程都是可见的，我们可以把这个分布式锁看作是一个资源，只有获取到资源的线程才可以继续向下执行（与单机程序中的锁很相似），那么如何让这个锁资源对不同节点的线程都可见呢?

那么就需要引入第三方来管理这个锁资源，如果有节点想要去获取锁，直接与这个第三方交互即可。至于锁状态值或者锁中数据的维护，都交由这个第三方来维护。节点只需要去请求下是否可以获取锁，然后这个第三方把结果告诉节点即可。这个第三方需要支持如果同时有三个请求想要获取锁，那么只能把这个锁给到其中的一个，其他两个请求要么阻塞，要么直接返回获取锁失败。

由此，我们可知这个管理节点的第三方应该是类似数据库，缓存这样的支持分布式系统的全局存储系统。

常见的分布式锁： 基于数据库，基于缓存，基于zookeeper，基于etcd等

分布式锁特性

考虑到JDK中锁的特性，我们大概可以得到分布式锁的特性如下：

1.互斥性：这个应该是分布式锁的基本特性，或者说也是锁的基本特性，保证资源（这里指共享代码段或者共享变量）同时只能有一个节点的某个线程访问；

2.可重入性：类似于ReentrantLock，同一服务节点的相同线程，允许重复多次加锁；

3.锁超时：特性与本地锁一致，一旦锁超时即释放拥有的锁资源；

4.非阻塞：支持获取锁的时候直接返回结果值（true or false），而不是在没有获取到锁的时候阻塞住线程的执行；

5.公平锁与非公平锁：公平锁是指按照请求加锁的顺序获得锁，非公平锁请求加锁是无序的。

以上的特性不是所有锁都需要支持的，根据具体的业务场景，可以只使用其中的某一种或者几种特性。

考虑到单点问题，这个分布式锁最好是集群模式，支持高可用，同时获取锁和释放锁的性能要好。

常见的分布式锁实现

- 基于MySQL数据库实现分布式锁

  > 基于MySql实现分布式锁有三种方式，分别是基于唯一索引，悲观锁与乐观锁(基于行锁)
  >
  > 基于MySql实现分布式锁主要是利用MySql自身的行锁机制，保证对于某一行，同时只能由一个线程对其查询或者更新或者插入（这就实现了分布式锁的最基本的特性：互斥性），而能够访问该行的线程即可以认为是获取到分布式锁的线程。

  1.基于唯一索引实现

  我们考虑一种最简单的MySQL实现分布式锁的方式：创建一个锁表，在表中添加一列名为lock_name，为这一列添加唯一索引unique key，对于某一个方法，如果有多个节点的多个线程同时访问，那么让这些线程去执行insert，由于唯一索引的存在，只会有一个线程插入成功，那么这个插入成功的线程就可以认为是获取到分布式锁的线程。

  > CREATE TABLE lock(
  > `id` int(11) NOT NULL AUTO_INCREMENT COMMENT '主键',
  > lock_name varchar(64) NOT NULL DEFAULT '' COMMENT '锁名',
  > `desc` varchar(1024) NOT NULL DEFAULT '备注信息',
  > `update_time` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP COMMENT '保存数据时间，自动生成',
  >
  > PRIMARY KEY (`id`),
  > UNIQUE KEY `uidx_lock_name` (`lock_name`) USING BTREE
  >) ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='锁定中的方法';
  >
  >
  > 获取锁执行insert语句，如果插入成功则说明可以成功获取到锁
  >
  > `insert into lock(lock_name,desc) values (‘lock_name’,‘desc’)`
  >
  > 释放锁则执行delete语句
  >
  > `delete from lock where lock_name='lock_name'`

  拓展：

  insert为非阻塞的，一旦插入失败就返回结果了，如果想要实现阻塞可以使用while循环；

要实现公平锁，可以引入一张表，记录因为插入失败而阻塞的线程，一旦锁被释放，被阻塞的线程可以根据插入的先后顺序来决定自己是否可以获取锁;

要实现可重入性，需要在锁表中增加一列，记录获取锁的服务节点信息与线程信息，获取锁的时候先查询数据库，如果当前机器的主机信息和线程信息在数据库可以查到的话，直接把锁分配给他就可以了;

要实现锁超时，需要在锁表增加一列，记录锁失效的时间，同时增加一个定时任务系统，定时扫描锁表中超时的记录，删除该条记录从而释放锁。

2.基于数据库排他锁实现

基于数据库排他锁for update实现，使用select ……. for update成功的线程可以获取到锁，失败的线程会被阻塞。而事务执行完毕，commit事务的时候就相当于释放了锁资源。

```
@Transaction
public void lock(String lockName) {
ResourceLock rlock = exeSql("select * from resource_lock where
resource_name = lockName for update");
if (rlock == null) {
      exeSql("insert into resource_lock(reosurce_name,owner) values
(lockName, 'ip')");
}
/**
 业务逻辑
**/
// 最后事务执行完毕，提交的时候就表示锁可以释放了
}
```

> 查询和插入的操作放在一个事务之中，事务开始执行select … for update查询的时候，代表有请求进来想要获取锁，如果查询成功或者发现锁表中没有该条记录，那么说明当前的请求可以获取该分布式锁资源。获取到锁资源之后，执行业务逻辑，业务逻辑执行完毕，事务提交，那么锁资源就被释放了。

MySql Innodb引擎中可以通过设置innodb_lock_wait_timeout控制全局的锁超时时间，默认是50s。

排他锁需要事务资源，会占用数据库连接，因为每个请求过来都会去for update，并且只有一个线程请求可以真正执行，其他的请求会持有连接资源阻塞住等待已经获取锁的线程释放。这样会造成大量的连接被占用，产生连接爆满的问题。

显然，使用排他锁的话很难实现锁重入。并且由于innodb_lock_wait_timeout是针对全局的，所以对锁超时的支持也不太好。

3.基于数据库乐观锁实现

我们知道，mysql乐观锁机制是在表中增加了version列，记录版本号。之后先查询某一行数据将版本号取出。更新的时候对版本号进行比较，一致的话，说明没有其他请求修改该条记录，那么就可以执行更新操作，否则的话无法更新。

```
select * from resource where resource_name = xxx
update resource set version= 'newVersion' ... where resource_name = xxx
and version = 'oldVersion
```

很显然，同一时刻只能有一个请求成功更新该条记录，那么我们就可以把成功更新该条记录的线程作为获取到分布式锁的线程。

```
@Transaction
public void lock(String lockName) {
```

```
ResourceLock rlock = exeSql("select * from resource_lock where
resource_name = lockName");
if (rlock == null) {
      exeSql("insert into resource_lock(reosurce_name,owner,version)
values (lockName, 'ip',1)");
}
/**
 业务逻辑
**/
Integer count = exeSql("update resource_lock set version=version+1
where  resource_name = lockName and version=#{version}");
if(count == 1) {
   // 成功获取到分布式锁，可以提交本次事务
} else {
   // 使用while重复执行select，update来获取分布式锁资源或者直接回滚本地事务
}

}
```

乐观锁的实现要先执行select再执行update，如果冲突比较多的话，大量的线程请求会执行CAS耗费CPU时间。

总结：基于数据库实现分布式锁，主要就是利用数据库自身的锁机制，保证插入，查询或者更新的排他性，但是由于数据库本身锁的开销以及性能，这种使用场景并不太多。

- 基于Redis实现分布式锁

> Redis除了用作缓存之外，也可以用来作为分布式锁。利用redis天然的支持分布式系统的特性以及某些命令的使用，可以较好的来实现分布式锁。redis的读写性能比较数据库来说也有极大的提升，目前也是一种较为流行的分布式锁解决方案。

1.SetNX命令实现分布式锁

> **SETNX key value**　将key的值设置为value，当且仅当key不存在的时候
>
> SETNX 是『SET if Not eXists』(如果不存在，则 SET)的简写

redis 2.6.12之前的版本是使用setnx+expire命令来实现分布式锁，expire命令是给锁设置过期时间避免发生死锁。

```
public static boolean lock(Jedis jedis, String lockKey, String
requestId, int expireTime) {
    // 使用requestId作为value是防止出现线程B把线程A的锁给释放了
   Long result = jedis.setnx(lockKey, requestId);
   //设置锁
   if (result == 1) {
       //获取锁成功
       //若在这里程序突然崩溃，则无法设置过期时间，将发生死锁
       //通过过期时间删除锁
       jedis.expire(lockKey, expireTime);
       return true;
   }
   return false;
}
```

由于上面setnx命令与expire命令非原子性，那么在获取锁成功准备去设置锁超时时间的时候，极端情况下系统崩溃了，那么该锁就没有设置超时时间，无法释放锁资源。

为了防止线程B把线程A的锁给释放了，设置value为当前客户端ID+线程ID作为唯一序列号，释放锁的时候需要判断key对应的value是否为加锁时候的value。

为了解决上面的问题，有两种方案：

1）方式一：lua脚本

既然是由于setnx与expire命令非原子性导致的，那么我们可以使用lua脚本来实现将这两个命令设置为原子性的。

```
// 加锁脚本，KEYS[1] 要加锁的key，ARGV[1]是UUID随机值，ARGV[2]是过期时间
private static final String SCRIPT_LOCK = "if redis.call('setnx',
KEYS[1], ARGV[1]) == 1 then redis.call('pexpire', KEYS[1], ARGV[2])
return 1 else return 0 end";

// 解锁脚本，KEYS[1]要解锁的key，ARGV[1]是UUID随机值
private static final String SCRIPT_UNLOCK = "if redis.call('get',
KEYS[1]) == ARGV[1] then return redis.call('del', KEYS[1]) else return
0 end";
```

2）方式一：设置value为锁过期时间

```
public static boolean wrongGetLock2(Jedis jedis, String lockKey, int
expireTime) {
long expires = System.currentTimeMillis() + expireTime;  // 设置锁的过期时
间
String expiresStr = String.valueOf(expires);

// 如果当前锁不存在，返回加锁成功
if (jedis.setnx(lockKey, expiresStr) == 1) {
   return true;
}

// 如果锁存在，获取锁的过期时间
String currentValueStr = jedis.get(lockKey);
if (currentValueStr != null && Long.parseLong(currentValueStr) <
System.currentTimeMillis()) {
   // 锁已过期，获取上一个锁的过期时间，并设置现在锁的过期时间
   String oldValueStr = jedis.getSet(lockKey, expiresStr);
   if (oldValueStr != null && oldValueStr.equals(currentValueStr)) {
       // 考虑多线程并发的情况，只有一个线程的设置值和当前值相同，它才有权利加锁
       return true;
   }
}

// 其他情况，一律返回加锁失败
return false;
}

// 释放锁
public static void wrongReleaseLock1(Jedis jedis, String lockKey) {
jedis.del(lockKey);
}
```

假设锁过期了，这个时候多个线程来请求获取锁，执行到jedis.getSet(lockKey, expiresStr)方法的时候，只有一个线程获取到的旧值是与原来的currentValueStr相同的，因为set会更新对应的lockKey的值，这之后的线程再调用getSet方法的时候获取到的值与currentValueStr就不同了。当然，这也会导致lockKey的值被覆盖。除了被覆盖的问题，还有就是各个客户端自己生成过期时间，而保证不同客户端时间的同步也是一个问题。还有就是这种情况下的分布式锁不具有标识，无法区分是哪个客户端加的锁，可能导致客户端A加的锁被客户端B释放了。

> 客户端A获取到分布式锁执行自己的程序，正常情况下，程序执行完毕需要释放锁资源，但是在某次程序执行的时候，由于调用第三方服务超时了，在没有执行到程序结束的时候，锁就已经过有效期了，这个时候客户端B就可以获取锁资源去执行自己的程序，在客户端B执行的过程中，客户端A执行的程序结束了，这个时候客户端A仍然执行释放锁的操作，结果就会把客户端B的锁给释放了。

2.SET key value [EX seconds] [PX milliseconds] [NX|XX]命令实现分布式锁

> **SET key value [EX seconds] [PX milliseconds] [NX|XX]**
>
> **可选参数**
>
> 从 Redis 2.6.12 版本开始， *SET* 命令的行为可以通过一系列参数来修改：
>
> - `EX second` ： 设置键的过期时间为 `second` 秒。 `SET key value EX second` 效果等同于 `SETEX key second value` 。
> - `PX millisecond` ： 设置键的过期时间为 `millisecond` 毫秒。 `SET key value PX millisecond` 效果等同于 `PSETEX key millisecond value` 。
> - `NX` ： 只在键不存在时，才对键进行设置操作。 `SET key value NX` 效果等同于 `SETNX key value` 。
> - `XX` ： 只在键已经存在时，才对键进行设置操作。

通过设置EX PX NX 这三个参数，可以实现setnx与expire命令的结合。这个是Redis 2.6.12 版本之后提供的原子性的命令。

```
// 获取锁
public static boolean tryGetDistributedLock(String lockKey,
   String requestId, int expireTime) {
Jedis jedis = RedisUtil.getJedis();
String result = jedis.set(lockKey, requestId, "NX", "PX", expireTime);
if ("OK".equals(result)) {
   return true;
}
return false;
}

// 释放锁，使用lua脚本
public static boolean tryGetDistributedLock(String lockKey,
   String requestId, int expireTime) {
Jedis jedis = RedisUtil.getJedis();
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)); // 执行lua脚本
if (1L.equals(result)) {
   return true;
}
```

```
return false;
}
```

3.Redission分布式锁

Redission是基于redis实现的分布式锁，主要使用的是redis的hash结构

> **HSET key field value**
>
> 以锁作为key，以客户端id+线程id作为作为field，以重入次数作为value
>
> 将哈希表 `key` 中的域 `field` 的值设为 `value` 。
>
> 如果 `key` 不存在，一个新的哈希表被创建并进行 HSET 操作。
>
> 如果域 `field` 已经存在于哈希表中，旧值将被覆盖。
>
> **HINCRBY key field increment**
>
> 为哈希表 `key` 中的域 `field` 的值加上增量 `increment` 。
>
> 增量也可以为负数，相当于对给定域进行减法操作。
>
> **HEXISTS key field**
>
> 查看哈希表 `key` 中，给定域 `field` 是否存在。

加锁lua源码

> KEYS[1]:表示你加锁的那个key
>
> ARGV[1]:表示锁的有效期，默认30s
> ARGV[2]:表示表示加锁的客户端ID+线程ID,类似于下面这样：
> 8743c9c0-0795-4907-87fd-6c719a6b4586:1
>
> // 首先判断锁key是否存在，如果不存在说明没有线程占用这个锁，你就可以加锁
>
> if (redis.call('exists', KEYS[1]) == 0) then
>
> // 加锁的话就是设置key的filed域为客户端ID:线程ID，设置value值为1
> redis.call('hset', KEYS[1], ARGV[2], 1);
>
> // 设置key过期时间
>
> redis.call('pexpire', KEYS[1], ARGV[1]);
> return nil;
> end;
>
> // 如果锁被占用，判断占用锁的线程是否跟要获取锁的线程是同一个客户端的同一个线程
>
> if (redis.call('hexists', KEYS[1], ARGV[2]) == 1) then
>
> // 如果是的话，就把对应的value值加1，表示锁重入次数加1
> redis.call('hincrby', KEYS[1], ARGV[2], 1);
>
> // 更新锁的过期时间
>
> redis.call('pexpire', KEYS[1], ARGV[1]);
> return nil;
> end;
> return redis.call('pttl', KEYS[1]); // 如果没有获取到锁，那么返回锁的剩余时间

加锁的话很简单就是用的hset命令： hset myLock 8743c9c0-0795-4907-87fd-6c719a6b4586:1 1

此时的结构为：

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

为了实现可重入，需要根据获取锁的线程的客户端+线程ID与占有锁的filed域值比较，相同的话，说明是同一个线程再次请求，那么只要把value值加1即可。这就是可重入。

执行加1之后结构为：

```
myLock:
{
  8743c9c0-0795-4907-87fd-6c719a6b4586:1 2
}
```

解锁lua源码

```
KEYS[1]:表示你加锁的那个key

ARGV[1]:表示锁的有效期，默认30s
ARGV[2]:表示表示加锁的客户端ID+线程ID,类似于下面这样：
8743c9c0-0795-4907-87fd-6c719a6b4586:1


//首先判断要释放锁的线程是不是锁的占用者

if (redis.call('hexists', KEYS[1], ARGV[2]) == 0) then
return nil;
end;

// 如说是锁的占用者，那么将对应的value值减1

local counter = redis.call('hincrby', KEYS[1], ARGV[3], -1);
if (counter > 0) then

   // 如果减一之后的value值仍旧大于0，说明该线程暂时无法释放锁，那么只要更新过期时间即可

    redis.call('pexpire', KEYS[1], ARGV[2]);
    return 0;

// 如果减1之后等于0，那么说明该线程已经可以释放锁了，直接调用del删除该key值即可

else redis.call('del', KEYS[1]);
redis.call('publish', KEYS[2], ARGV[1]);
return 1;
end;
return nil;
```

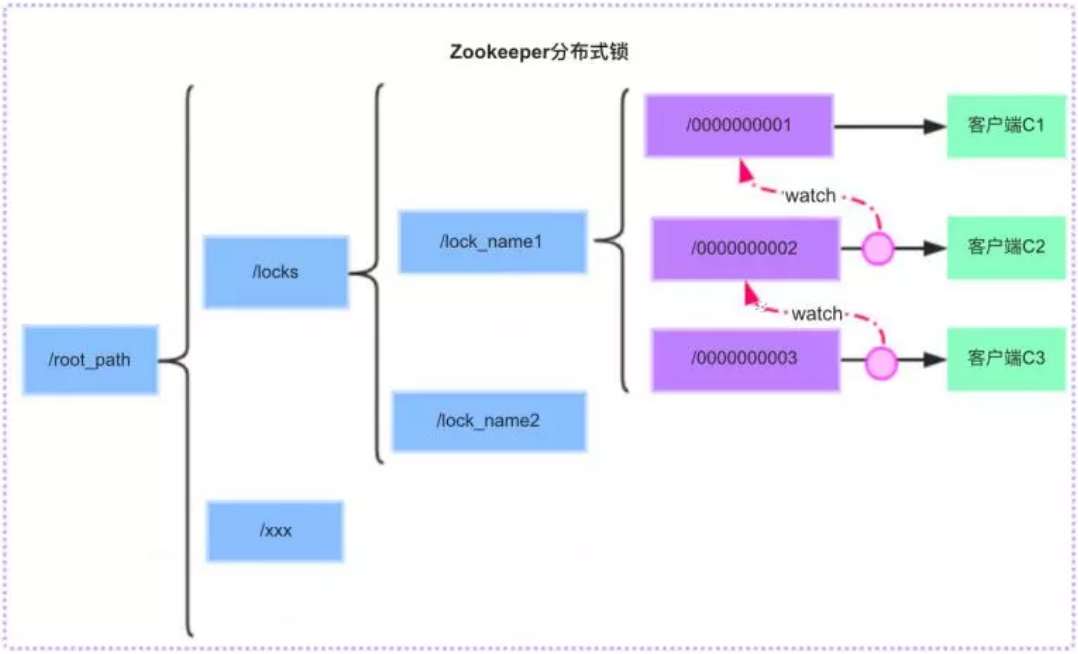
另外，Redisson提供了watch dog机制，可以在锁将要过期的时候自动延期，防止程序尚未执行完毕锁即被释放。

最大的问题，就是如果你对某个redis master实例，写入了myLock这种锁key的value，此时会异步复制给对应的master slave实例。但是这个过程中一旦发生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。这时系统在业务上一定会出现问题，导致脏数据的产生。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，可能导致多个客户端同时完成加锁。

- 基于Zookeeper实现分布式锁

> Zookeeper 是一种提供「分布式服务协调」的中心化服务，类似于Unix文件系统结构，可以看作一棵树，每个节点叫做ZNode。ZNode中可以存储数据。ZNode节点分为两种类型：
>
> 临时节点：当客户端和服务端断开连接后，所创建的Znode(节点)会自动删除
>
> 根据节点是否有序分为普通临时节点与有序临时节点
>
> 持久化节点：当客户端和服务端断开连接后，所创建的Znode(节点)不会删除
>
> 根据节点是否有序分为普通持久化节点与有序持久化节点
>
> Zookeeper还提供了Watcher机制，Zookeeper允许用户在指定节点上注册一些Watcher，并且在一些特定事件触发的时候(比如节点存储信息改变/节点被删除/节点新增（删除）子节点等)，ZooKeeper服务端会将事件通知到感兴趣的客户端上去，该机制是Zookeeper实现分布式协调服务的重要特性。



基于zookeeper的以上特性，可以实现分布式锁。

> 首先创建持久化节点/lock_name1，每当有客户端来访问/lock_name1节点，就在该节点下创建一个临时有序子节点，由于临时有序节点是递增的，所以总有一个临时有序节点的序号是最小的，那么这个序号最小的节点就可以获得分布式锁。而其他未获取到锁请求则通过Watcher机制监听上一个比自己序号小的节点。
>
> 1.客户端C1拿到/lock_name1下面所有的子节点，比较序号，发现自己序号最小，所以得到锁。
>
> 2.客户端C2拿到/lock_name1下面所有的子节点，比较序号，发现自己的序号不是最小的，所以客户端C2并不会获取到锁，而是监听比自己序号小的上一个节点C1的状态。
>
> 3.客户端C3拿到/lock_name1下面所有的子节点，比较序号，发现自己的序号不是最小的，所以客户端C3并不会获取到锁，而是监听比自己序号小的上一个节点C2的状态。
>
> ....
>
> n.客户端C1执行完毕，释放锁资源，同时C1节点被删除，而C2监控到C1节点状态发生变化，比较之后发现自己的序号最小，所以可以获取锁。

总结：zookeeper实现分布式锁主要也是使用了zookeeper的自身的特性，即使同一时刻多个请求过来，创建的节点的序号也是递增的且不会重复。而利用监听机制可以保证锁被释放之后其他节点可以获取到该信息从而有机会去获取锁。

- 基于ETCD实现分布式锁

> ETCD是一个高可用的分布式键值存储数据库，主要用于共享配置和服务发现。
>
> 1.lease功能，就是租约机制(time to live)。
>
> . etcd可以对存储key-value的数据设置租约，也就是给key-value设置一个过期时间，当租约到期，key-value将会失效而被etcd删除。
>
> . etcd同时也支持续约租期，可以通过客户端在租约到期之间续约，以避免key-value失效；
>
> . etcd还支持解约，一旦解约，与该租约绑定的key-value将会失效而删除。
>
> Lease 功能可以保证分布式锁的安全性，为锁对应的 key 配置租约，即使锁的持有者因故障而不能主动释放锁，锁也会因租约到期而自动释放。
>
> 2.watch功能
>
> 监听功能。watch 机制支持监听某个固定的key，它也支持watch一个范围（前缀机制），当被watch的key或范围发生变化时，客户端将收到通知。
>
> 在实现分布式锁时，如果抢锁失败，可通过 Prefix 机制返回的 KeyValue 列表获得 Revision 比自己小且相差最小的 key（称为 pre-key），对 pre-key 进行监听，因为只有它释放锁，自己才能获得锁，如果 Watch 到 pre-key 的 DELETE 事件，则说明pre-ke已经释放，自己已经持有锁。
>
> 3.prefix功能
>
> 前缀机制。也称目录机制，如两个 key 命名如下：key1="/mykey/key1"，key2="/mykey/key2"，那么，可以通过前缀-"/mykey"查询，返回包含两个 key-value 对的列表。可以和前面的watch功能配合使用。
>
> 例如，一个名为 /mylock 的锁，两个争抢它的客户端进行写操作，实际写入的 key 分别为：key1="/mylock/UUID1"，key2="/mylock/UUID2"，其中，UUID 表示全局唯一的 ID，确保两个 key 的唯一性。很显然，写操作都会成功，但返回的 Revision 不一样，那么，如何判断谁获得了锁呢？通过前缀 /mylock 查询，返回包含两个 key-value 对的的 KeyValue 列表，同时也包含它们的 Revision，通过 Revision 大小，客户端可以判断自己是否获得锁，如果抢锁失败，则等待锁释放（对应的 key 被删除或者租约过期），然后再判断自己是否可以获得锁。
>
> 4.revision功能
>
> Revision是针对全局的一个版本号，每进行一次事务，revision值会递增，比如初始的revision为0，执行put(key, value) key对应的revision变为1，执行put(key1, value)的话key1的revision变为2。
>
> 通过这种机制，可以知道key的写入顺序，在实现分布式锁时，多个客户端同时抢锁，根据key的revision大小排序，key对应的revision值最小的获取锁。

假设某个共享资源的锁名称为： /lock/mylock

第一步：客户端连接ETCD，以 /lock/mylock为前缀创建全局唯一的key，客户端一的key=/lock/mylock/UUID1，客户端二的key=/lock/mylock/UUID2，客户端....；客户端分别为自己的key设置租约(有效期)，防止因为etcd集群崩溃而key未被删除出现死锁；

第二步：客户端创建一个定时任务，定时检查key的有效期，防止其他未获取到锁的客户端的key过期。此外，如果持有锁期间客户端崩溃，心跳停止，key 将因租约到期而被删除，从而锁释放，避免死锁。

第三步：客户端将自己的key写入etcd，同时也要记录返回的revision；

第四步：客户端以前缀/lock/mylock读取value符合前缀的列表。取出所有的等待共享锁的客户端列表，判断自己的revision是不是列表中最小的，如果是的话，就认为自己可以获取锁，否则监听列表中前一个比自己revision值小的客户端，一旦监听到这个客户端的del事件，说明自己可以获取锁了。

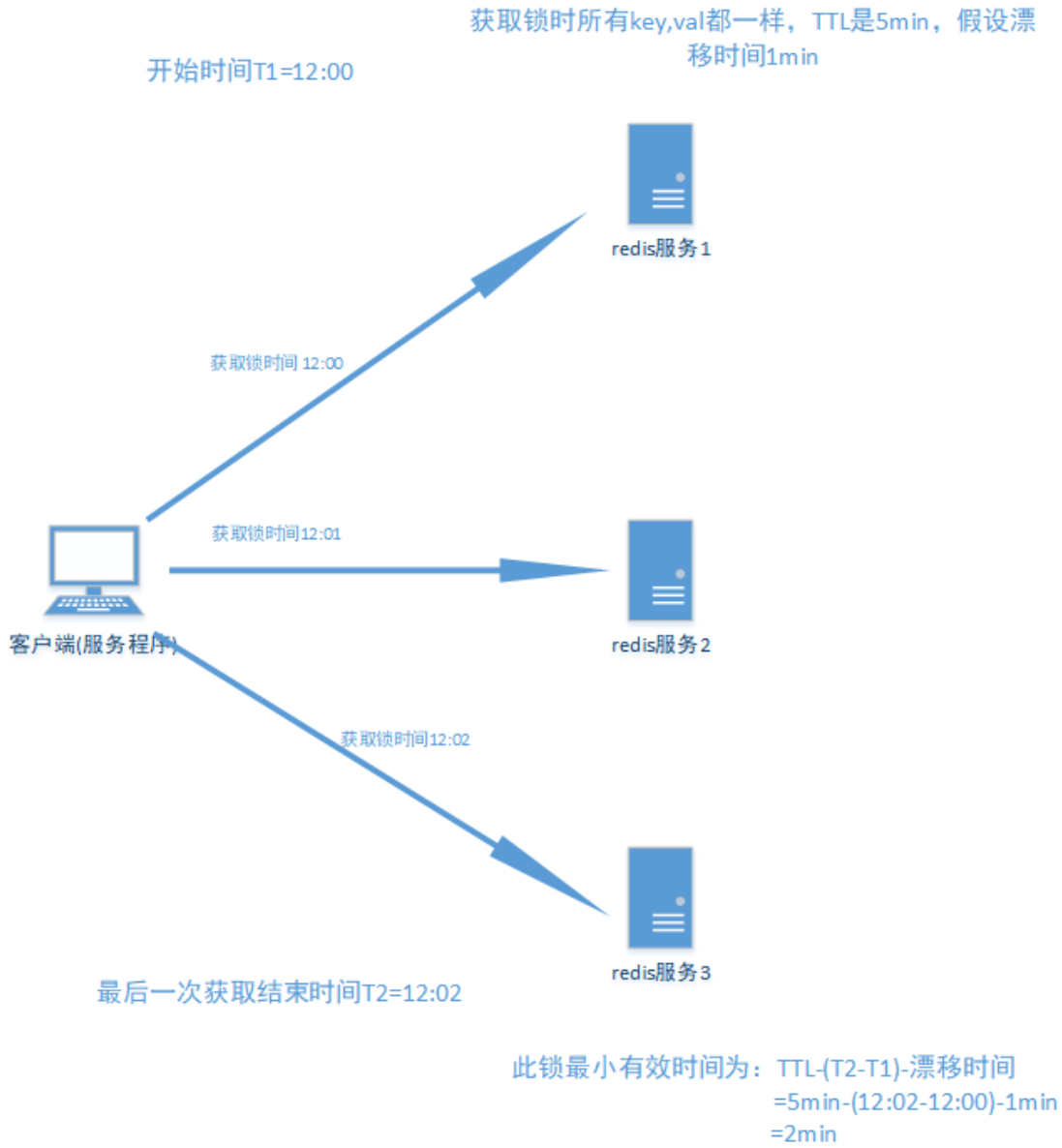
第五步：获取锁资源，执行业务逻辑

第六步：删除key，释放锁资源

> 值得注意的是不论是基于zookeeper实现的分布式锁还是基于ETCD实现的分布式锁，由于这两个分布式中间件都是CP系统，所以性能不如redis实现的分布式锁，但是在可靠性上要比redis实现的分布式锁要好，因为redis集群是基于AP系统的。

Redlock(红锁)

> **多节点redis实现的分布式锁算法(RedLock):有效防止单点故障**
>
> 假设有5个**完全独立**的redis主服务器
>
> 1.获取当前时间戳
>
> 2.client尝试按照顺序使用相同的key,value获取所有redis服务的锁，在获取锁的过程中的获取时间比锁过期时间短很多，这是为了不要过长时间等待已经关闭的redis服务。并且试着获取下一个redis实例。
>
> 比如：TTL为5s,设置获取锁最多用1s，所以如果一秒内无法获取锁，就放弃获取这个锁，从而尝试获取下个锁
>
> 3.client通过获取所有能获取的锁后的时间减去第一步的时间，这个时间差要小于TTL时间并且至少有3个redis实例成功获取锁，才算真正的获取锁成功
>
> 4.如果成功获取锁，则锁的真正有效时间是 TTL减去第三步的时间差 的时间；比如：TTL 是5s,获取所有锁用了2s,则真正锁有效时间为3s(其实应该再减去时钟漂移);
>
> 5.如果客户端由于某些原因获取锁失败，便会开始解锁所有redis实例；因为可能已经获取了小于3个锁，必须释放，否则影响其他client获取锁
>
> 算法示意图如下：



总结:

1.TTL时长 要大于正常业务执行的时间+获取所有redis服务消耗时间+时钟漂移

2.获取redis所有服务消耗时间要 远小于TTL时间，并且获取成功的锁个数要 在总数的一般以上:N/2+1

3.尝试获取每个redis实例锁时的时间要 远小于TTL时间

4.尝试获取所有锁失败后 重新尝试一定要有一定次数限制

5.在redis崩溃后（无论一个还是所有），要延迟TTL时间重启redis

6.在实现多redis节点时要结合单节点分布式锁算法 共同实现

参考:

https://blog.csdn.net/u013000898/article/details/87904591 etcd实现分布式锁流程

https://www.cnblogs.com/jiujuan/p/12147809.html etcd实现分布式锁分析

https://www.hollischuang.com/archives/1716 分布式锁的几种实现方式

https://www.cnblogs.com/kiko2014551511/p/11527108.html Redission分布式锁原理