

# 零声教育出品 Mark 老师

## QQ:2548898954

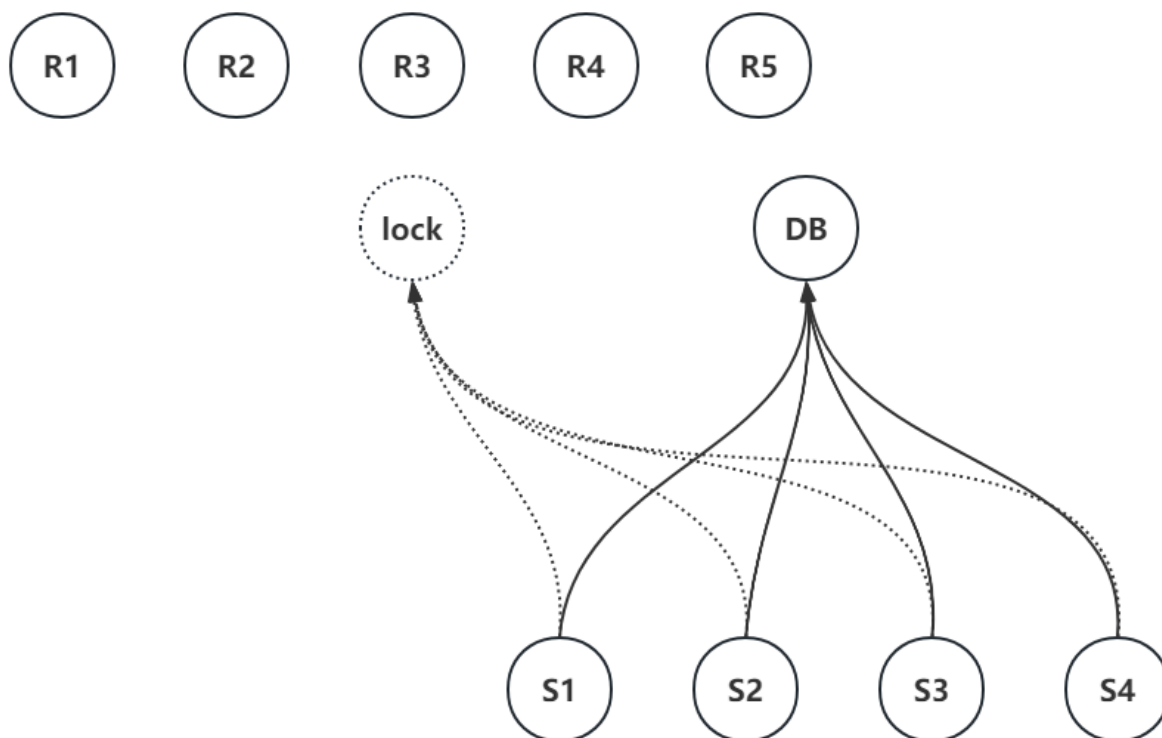
---

## 背景

---

在分布式系统中，一个应用部署在多台机器当中，在某些场景下，为了保证数据一致性，要求在同一时刻，同一任务只在一个节点上运行，即保证某个行为在同一时刻只能被一个线程执行；在单机单进程多线程环境，通过锁很容易做到，比如 *mutex*、*spinlock*、信号量等；而在多机多进程环境中，此时就需要分布式锁来解决了；

```
1 // 互斥锁的使用：
2 pthread_mutex_init(&mutex, NULL);
3
4 pthread_mutex_lock(&mutex);
5 // ....
6 pthread_mutex_unlock(&mutex);
7
8 pthread_mutex_destroy(&mutex);
```



## 常见实现方式

- 基于数据库
- 基于缓存 *redis*
- 基于 *zk / etcd*

## 接口实现

- 加锁
- 解锁

## 注意事项

- 互斥性

同时只允许一个持锁对象进入临界资源；其他待持锁对象要么等待，要么轮询检测是否能获取锁；

- 锁超时

允许持锁对象持锁最长时间；如果持锁对象宕机，需要外力解除锁定，方便其他持锁对象获取锁；

- 高可用

锁存储位置若宕机，可能引发整个系统不可用；应有备份存储位置和切换备份存储的机制，从而确保服务可用；

- 容错性

若锁存储位置宕机，恰好锁丢失的话，是否能正确处理；

## 类型

---

- 重入锁和非重入锁；

是否允许持锁对象再次获取锁；

- 公平锁和非公平锁；

如果同时争夺锁是否获取锁的几率一样？

公平锁通常通过排队来实现；

非公平锁通常不间断尝试获取锁来实现；

## MySQL 实现分布式锁

---

主要利用 MySQL 唯一键的唯一性约束来实现互斥性；

## 表结构

```
1 DROP TABLE IF EXISTS `dislock`;
2 CREATE TABLE `dislock` (
3   `id` int(11) unsigned NOT NULL AUTO_INCREMENT
  COMMENT '主键',
4   `lock_type` varchar(64) NOT NULL COMMENT '锁类
  型',
5   `owner_id` varchar(255) NOT NULL COMMENT '持锁对
  象',
6   `update_time` timestamp NOT NULL DEFAULT
  CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
7   PRIMARY KEY (`id`),
8   UNIQUE KEY `idx_lock_type` (`lock_type`)
9 ) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT
  CHARSET=utf8 COMMENT='分布式锁表';
```

## 加锁

```
1 INSERT INTO dislock (`lock_type`, `owner_id`)
  VALUES ('act_lock', 'ad2daf3');
```

## 解锁

### 进程解锁

```
1 DELETE FROM dislock WHERE `lock_type` = 'act_lock'
  AND `owner_id` = 'ad2daf3';
```

### 超进程解锁

```
1 delete from dislock where `lock_type` =
  'act_lock';
```

# 总结

- 可用性依赖数据库；若数据库是单点，挂掉将导致业务系统不可用；
- 还需额外实现锁失效的问题；解锁失败，其他线程将无法获得锁；

## redis 实现非公平锁

### 分布式锁流程

- 尝试获取锁；
  - 获取锁成功，操作临界资源，操作结束尝试释放锁；
  - 获取锁失败，订阅解锁信息；
- 尝试释放锁；
  - 释放锁成功，广播解锁信息；
  - 释放锁失败（说明此时持有锁对象不是自己），获得锁失效时间；

## 加锁

```
1  --[[
2      KEYS[1]          lock_name
3      KEYS[2]          lock_channel_name
4      ARGV[1]          lock_time (ms)
5      ARGV[2]          uuid
6  ]]
7  if redis.call('exists', KEYS[1]) == 0 then
8      redis.call('hset', KEYS[1], ARGV[2], 1)
9      redis.call('pexpire', KEYS[1], ARGV[1])
10     return
11 end
12 -- 若支持锁重入，将注释去掉
```

```

13 -- if redis.call('hexists', KEYS[1], ARGV[2]) ==
    1 then
14 --     redis.call('hincrby', KEYS[1], ARGV[2], 1)
15 --     redis.call('pexpire', KEYS[1], ARGV[1])
16 --     return
17 -- end
18 redis.call("subscribe", KEYS[2])
19 return redis.call('pttl', KEYS[1])

```

## 解锁

```

1 --[[
2     KEYS[1]          lock_name
3     KEYS[2]          uuid
4 ]]
5 local uuid = redis.call("get", KEYS[1])
6 if uuid == KEYS[2] then
7     redis.call("del", KEYS[1])
8 end

```

```

1 --[[
2     KEYS[1]          lock_name
3     KEYS[2]          lock_channel_name
4     ARGV[1]          0 sign of unlock
5     ARGV[2]          lock_expire_time
6     ARGV[3]          uuid
7 ]]
8 if redis.call('exists', KEYS[1]) == 0 then
9     redis.call('publish', KEYS[2], ARGV[1])
10    return 1
11 end
12
13 if redis.call('hexists', KEYS[1], ARGV[3]) == 0
    then
14    return

```

```

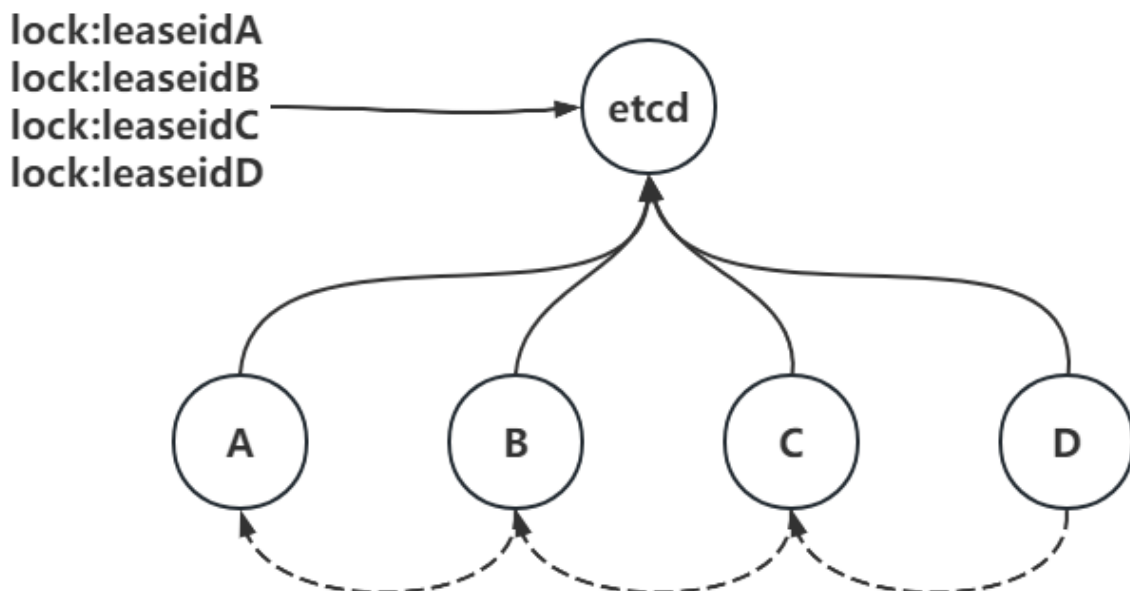
15 end
16
17 -- 若支持锁重入，将注释去掉
18 -- local cnt = redis.call('hincrby', KEYS[1],
    ARGV[3], -1)
19 -- if cnt > 0 then
20 --     redis.call('pexpire', KEYS[1], ARGV[2])
21 --     return 0
22 -- else
23     redis.call('del', KEYS[1])
24     redis.call('publish', KEYS[2], ARGV[1])
25     return 1
26 -- end

```

## 总结

- 可用性依赖 redis 的可用性;
- 容错性很差, redis 采用的异步复制, 数据可能丢失;
- 效率最高的一种分布式锁;
- 最安全的 redis 分布式锁: <https://github.com/jacket-code/redlock-cpp.git>;
- <http://redis.cn/topics/distlock.html>

## etcd 实现公平锁



# 数据版本号机制

- term:  
leader 任期, leader 切换时 term 加一; 全局单调递增, 64bits;
- revision:  
etcd 键空间版本号, key 发生变更, 则 revision 加一; 全局单调递增, 64bits;
- kvs:
  - create\_revision  
创建数据时, 对应的版本号;
  - mod\_revision  
数据修改时, 对应的版本号;
  - version  
当前的版本号; 标识该 val 被修改了多少次;

**注意:** 版本号机制在分布式锁中的作用为**避免重复加锁**同时实现**按插入顺序排序**;

## 租约

用于集群监控以及服务注册发现;

**注意:** 在分布式锁中作用为实现**全局唯一 id** 并且**实现锁失效**;



```

1 etcdctl lease grant <ttl> [flags]          创建
   一个租约
2 etcdctl lease keep-alive [options] <leaseID>
   [flags]          续约
3 etcdctl lease list [flags]
   枚举所有的租约
4 etcdctl lease revoke <leaseID> [flags]
   销毁租约
5
6 etcdctl lease timetolive <leaseID> [options]
   [flags]          获取租约信息
7 OPTIONS:
8     --keys[=false]    Get keys attached to this
   lease

```

## 事务

```

1 NAME:
2     txn - Txn processes all the requests in
   one transaction
3
4 USAGE:
5     etcdctl txn [options] [flags]
6
7 OPTIONS:
8     -h, --help[=false]    help for txn
9     -i, --interactive[=false]    Input transaction
   in interactive mode
10
11 事务
12 1. 比较
13     1. 比较运算符 > = < !=
14     2. create 获取key的create_revision
15     3. mod 获取key的mod_revision
16     4. value 获取key的value
17     5. version 获取key的修改次数

```

```
18 2. 比较成功
19   1. 成功后可以操作多个 del put get
20   2. 这些操作保证原子性
21 3. 比较失败
22   1. 成功后可以操作多个 del put get
23   2. 这些操作保证原子性
```

用于分布式锁以及leader选举；**保证多个操作的原子性**；确保多个节点数据读写的一致性；

```
TXN if/ then/ else ops
```

## 加锁

```
1 func (m *Mutex) Lock(ctx context.Context) error {
2     s := m.s
3     client := m.s.Client()
4
5     m.myKey = fmt.Sprintf("%s%x", m.pfx, s.Lease())
6     cmp := v3.Compare(v3.CreateRevision(m.myKey),
7         "=", 0)
8     // put self in lock waiters via myKey; oldest
9     // waiter holds lock
10    put := v3.OpPut(m.myKey, "",
11        v3.WithLease(s.Lease()))
12    // reuse key in case this session already holds
13    // the lock
14    get := v3.OpGet(m.myKey)
15    // fetch current holder to complete uncontended
16    // path with only one RPC
17    getOwner := v3.OpGet(m.pfx,
18        v3.WithFirstCreate()...)
19    resp, err := client.Txn(ctx).If(cmp).Then(put,
20        getOwner).Else(get, getOwner).Commit()
21    if err != nil {
22        return err
23    }
24 }
```

```

17     m.myRev = resp.Header.Revision
18     if !resp.Succeeded {
19         m.myRev =
20         resp.Responses[0].GetResponseRange().Kvs[0].CreateRevision
21     }
22     // if no key on prefix / the minimum rev is
23     key, already hold the lock
24     ownerKey :=
25     resp.Responses[1].GetResponseRange().Kvs
26     if len(ownerKey) == 0 ||
27     ownerKey[0].CreateRevision == m.myRev {
28         m.hdr = resp.Header
29         return nil
30     }
31     // wait for deletion revisions prior to myKey
32     hdr, werr := waitDeletes(ctx, client, m.pfx,
33     m.myRev-1)
34     // release lock key if wait failed
35     if werr != nil {
36         m.Unlock(client.Ctx())
37     } else {
38         m.hdr = hdr
39     }
40     return werr
41 }
42
43 // waitDeletes efficiently waits until all keys
44 matching the prefix and no greater
45 // than the create revision.
46 func waitDeletes(ctx context.Context, client
47 *v3.Client, pfx string, maxCreateRev int64)
48 (*pb.ResponseHeader, error) {
49     getOpts := append(v3.WithLastCreate(),
50     v3.WithMaxCreateRev(maxCreateRev))
51     for {

```

```

44     resp, err := client.Get(ctx, pfx, getOpts...)
45     if err != nil {
46         return nil, err
47     }
48     if len(resp.Kvs) == 0 {
49         return resp.Header, nil
50     }
51     lastKey := string(resp.Kvs[0].Key)
52     if err = waitDelete(ctx, client, lastKey,
resp.Header.Revision); err != nil {
53         return nil, err
54     }
55 }
56 }

```

## 解锁

```

1 func (m *Mutex) Unlock(ctx context.Context) error
{
2     client := m.s.Client()
3     if _, err := client.Delete(ctx, m.myKey); err !=
nil {
4         return err
5     }
6     m.myKey = "\x00"
7     m.myRev = -1
8     return nil
9 }

```