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；

```
// 互斥锁的使用：
pthread_mutex_init(&mutex, NULL);

pthread_mutex_lock(&mutex);
// ....
pthread_mutex_unlock(&mutex);

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

> 琐存储位置若宕机，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

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

加锁

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

解锁

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

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

```
--[[
    KEYS[1]         lock_name
    KEYS[2]         lock_channel_name
    ARGV[1]         lock_time (ms)
    ARGV[2]         uuid
]]
if redis.call('exists', KEYS[1]) == 0 then
    redis.call('hset', KEYS[1], ARGV[2], 1)
    redis.call('pexpire', KEYS[1], ARGV[1])
    return
end
-- 若支持锁重入，将注释去掉
-- 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
-- end
redis.call("subscribe", KEYS[2])
return redis.call('pttl', KEYS[1])
```

解锁

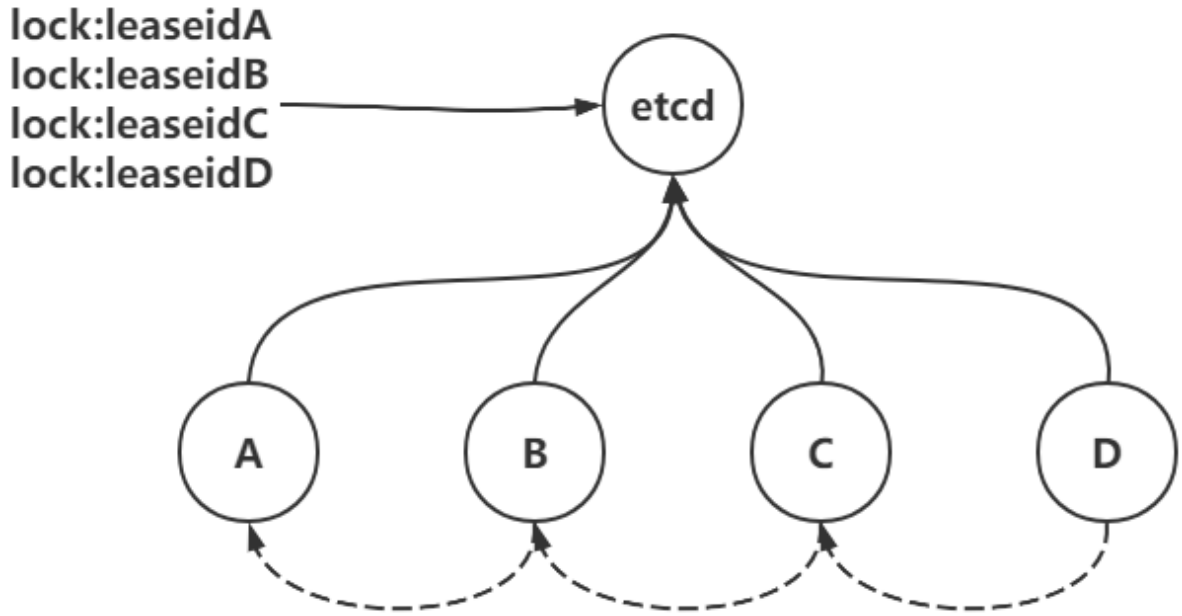
```
--[[
    KEYS[1]         lock_name
    KEYS[2]         lock_channel_name
    ARGV[1]         0 sign of unlock
    ARGV[2]         lock_expire_time
    ARGV[3]         uuid
]]
if redis.call('exists', KEYS[1]) == 0 then
    redis.call('publish', KEYS[2], ARGV[1])
    return 1
end

if redis.call('hexists', KEYS[1], ARGV[3]) == 0 then
    return
end

-- 若支持锁重入，将注释去掉
-- local cnt = redis.call('hincrby', KEYS[1], ARGV[3], -1)
-- if cnt > 0 then
--     redis.call('pexpire', KEYS[1], ARGV[2])
--     return 0
-- else
    redis.call('del', KEYS[1])
    redis.call('publish', KEYS[2], ARGV[1])
    return 1
-- 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；

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**；

```
etcdctl lease grant <ttl> [flags]
创建一个租约
etcdctl lease keep-alive [options] <leaseID>
[flags]         续约
etcdctl lease list [flags]
枚举所有的租约
etcdctl lease revoke <leaseID> [flags]
          销毁租约

etcdctl lease timetolive <leaseID> [options]
[flags]         获取租约信息
OPTIONS:
      --keys[=false]    Get keys attached to this
lease
```

事务

```
NAME:
        txn - Txn processes all the requests in
one transaction

USAGE:
        etcdctl txn [options] [flags]

OPTIONS:
  -h, --help[=false]            help for txn
```

```
  -i, --interactive[=false]      Input transaction in interactive mode

事务
1. 比较
    1. 比较运算符 > = < !=
    2. create 获取key的create_revision
    3. mod 获取key的mod_revision
    4. value 获取key的value
    5. version 获取key的修改次数
2. 比较成功
    1. 成功后可以操作多个 del put get
    2. 这些操作保证原子性
3. 比较失败
    1. 成功后可以操作多个 del put get
    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

```
func (m *Mutex) Lock(ctx context.Context) error {
    s := m.s
    client := m.s.Client()

    m.myKey = fmt.Sprintf("%s%x", m.pfx, s.Lease())
    cmp := v3.Compare(v3.CreateRevision(m.myKey), "=", 0)
    // put self in lock waiters via myKey; oldest waiter holds lock
    put := v3.OpPut(m.myKey, "", v3.WithLease(s.Lease()))

```

```
    // reuse key in case this session already holds the lock
    get := v3.OpGet(m.myKey)
    // fetch current holder to complete uncontended path with only one RPC
    getOwner := v3.OpGet(m.pfx, v3.WithFirstCreate()...)
    resp, err := client.Txn(ctx).If(cmp).Then(put, getOwner).Else(get, getOwner).Commit()
    if err != nil {
        return err
    }
    m.myRev = resp.Header.Revision
    if !resp.Succeeded {
        m.myRev = resp.Responses[0].GetResponseRange().Kvs[0].CreateRevision
    }
    // if no key on prefix / the minimum rev is key, already hold the lock
    ownerKey := resp.Responses[1].GetResponseRange().Kvs
    if len(ownerKey) == 0 || ownerKey[0].CreateRevision == m.myRev {
        m.hdr = resp.Header
        return nil
    }

    // wait for deletion revisions prior to myKey
    hdr, werr := waitDeletes(ctx, client, m.pfx, m.myRev-1)
    // release lock key if wait failed
    if werr != nil {
        m.Unlock(client.Ctx())
    } else {
        m.hdr = hdr
```

```
35      }
36      return werr
37  }
38
39  // waitDeletes efficiently waits until all keys
    matching the prefix and no greater
40  // than the create revision.
41  func waitDeletes(ctx context.Context, client
    *v3.Client, pfx string, maxCreateRev int64)
    (*pb.ResponseHeader, error) {
42      getOpts := append(v3.WithLastCreate(),
    v3.WithMaxCreateRev(maxCreateRev))
43      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

```go
func (m *Mutex) Unlock(ctx context.Context) error {
    client := m.s.Client()
    if _, err := client.Delete(ctx, m.myKey); err != nil {
        return err
    }
    m.myKey = "\x00"
    m.myRev = -1
    return nil
}
```