林湾村龙猫

2018年04月14日 阅读 1787

6

【180414】分布式锁（redis/mysql）

单台机器所能承载的量是有限的，用户的量级上万，基本上服务都会做分布式集群部署。很多时候，会遇到对同一资源的方法。这时候就需要锁，如果是单机版的，可以利用java等语言自带的并发同步处理。如果是多台机器部署就得要有个中间代理人来做分布式锁了。

常用的分布式锁的实现有三种方式。

- 基于redis实现（利用redis的原子性操作setnx来实现）
- 基于mysql实现（利用mysql的innodb的行锁来实现，有两种方式， 悲观锁与乐观锁）
- 基于Zookeeper实现（利用zk的临时顺序节点来实现）

目前，我已经是用了redis和mysql实现了锁，并且根据应用场景应用在不同的线上环境中。zk实现比较复杂，又无应用场景，有兴趣的可以参考他山之石中的《Zookeeper实现分布式锁》。

说说心得和体会。

> 没有什么完美的技术、没有万能钥匙、不同方式不同应用场景 CAP原理：一致性（consistency）、可用性（availability）、分区可容忍性（partition-tolerance）三者取其二。

他山之石

- Zookeeper实现分布式锁：www.jianshu.com/p/5d12a0101…
- 分布式锁的几种实现方式~：www.hollischuang.com/archives/17…
- select for update引发死锁分析:www.cnblogs.com/micrari/p/8…

基于redis缓存实现分布式锁

基于redis的锁实现比较简单，由于redis的执行是单线程执行，天然的具备原子性操作，我们可以利用命令setnx和expire来实现，java版代码参考如下：

java 复制代码

```java
package com.fenqile.creditcard.appgatewaysale.provider.util;

import com.fenqile.redis.JedisProxy;

import java.util.Date;

/**
 * User: Rudy Tan
 * Date: 2017/11/20
 *
 * redis 相关操作
 */
public class RedisUtil {

    /**
     * 获取分布式锁
     *
     * @param key         string 缓存key
```

```
     * @param expireTime int 过期时间，单位秒
     * @return boolean true-抢到锁，false-没有抢到锁
     */
    public static boolean getDistributedLockSetTime(String key, Integer expireTime) {
        try {
            // 移除已经失效的锁
            String temp = JedisProxy.getMasterInstance().get(key);
            Long currentTime = (new Date()).getTime();
            if (null != temp && Long.valueOf(temp) < currentTime) {
                JedisProxy.getMasterInstance().del(key);
            }

            // 锁竞争
            Long nextTime = currentTime + Long.valueOf(expireTime) * 1000;
            Long result = JedisProxy.getMasterInstance().setnx(key, String.valueOf(nextTime));
            if (result == 1) {
                JedisProxy.getMasterInstance().expire(key, expireTime);
                return true;
            }
        } catch (Exception ignored) {
        }
        return false;
    }
}
```

包名和获取redis操作对象换成自己的就好了。

基本步骤是

1. 每次进来先检测一下这个key是否实现。如果失效了移除失效锁
2. 使用setnx原子命令争抢锁。
3. 抢到锁的设置过期时间。

步骤2为最核心的东西， 为啥设置步骤3？可能应为获取到锁的线程出现什么移除请求，而无法释放锁，因此设置一个最长锁时间，避免死锁。 为啥设置步骤1？redis可能在设置expire的时候挂掉。设置过期时间不成功，而出现锁永久生效。

线上环境，步骤1、3的问题都出现过。所以要做保底拦截。

redis集群部署

6 通常redis都是以master-slave解决单点问题，多个master-slave组成大集群，然后通过一致性哈希算法将不同的key路由到不同master-slave节点上。

redis锁的优缺点：

优点：redis本身是内存操作、并且通常是多片部署，因此有这较高的并发控制，可以抗住大量的请求。 缺点：redis本身是缓存，有一定概率出现数据不一致请求。

在线上，之前，利用redis做库存计数器，奖品发放理论上只发放10个的，最后发放了14个。出现了数据的一致性问题。

因此在这之后，引入了mysql数据库分布式锁。

基于mysql实现的分布式锁。

实现第一版

在此之前，在网上搜索了大量的文章，基本上都是 插入、删除发的方式或是直接通过"select for update"这种形式获取锁、计数器。具体可以参考他山之石中的《分布式锁的几种实现方式~》关于数据库锁章节。

一开始，我的实现方式伪代码如下：

```java
public boolean getLock(String key){
      select for update
      if （记录存在）{
            update
      }else {
            insert
    }
}
```

这样实现出现了很严重的死锁问题，具体原因可以可以参考他山之石中的《select for update引发死锁分析》 这个版本中存在如下几个比较严重的问题：

1.通常线上数据是不允许做物理删除的 2.通过唯一键重复报错，处理错误形式是不太合理的。 3.如果appclient在处理中还没释放锁之前就挂掉了，会出现锁一直存在，出现死锁。 4.如果以这种方式，实现redis中的计数器（incr decr）,当记录不存在的时候，会出现大量死锁的情况。

因此考虑引入，记录状态字段、中央锁概念。

实现第二版

在第二版中完善了数据库表设计，参考如下：

```sql
-- 锁表，单库单表
CREATE TABLE IF NOT EXISTS credit_card_user_tag_db.t_tag_lock (

    -- 记录index
```

```
    Findex INT NOT NULL AUTO_INCREMENT COMMENT '自增索引id',

    -- 锁信息（key、计数器、过期时间、记录描述）
    Flock_name VARCHAR(128) DEFAULT '' NOT NULL COMMENT '锁名key值',
    Fcount INT NOT NULL DEFAULT 0 COMMENT '计数器',
    Fdeadline DATETIME NOT NULL DEFAULT '1970-01-01 00:00:00' COMMENT '锁过期时间',
    Fdesc VARCHAR(255) DEFAULT '' NOT NULL COMMENT '值/描述',

    -- 记录状态及相关事件
    Fcreate_time DATETIME NOT NULL DEFAULT '1970-01-01 00:00:00' COMMENT '创建时间',
    Fmodify_time DATETIME NOT NULL DEFAULT '1970-01-01 00:00:00' COMMENT '修改时间',
    Fstatus TINYINT NOT NULL DEFAULT 1 COMMENT '记录状态，0：无效，1：有效',

    -- 主键（PS：总索引数不能超过5）
    PRIMARY KEY (Findex),
    -- 唯一约束
    UNIQUE KEY uniq_Flock_name(Flock_name),
    -- 普通索引
    KEY idx_Fmodify_time(Fmodify_time)

)ENGINE=INNODB DEFAULT CHARSET=UTF8 COMMENT '信用卡|锁与计数器表|rudytan|20180412';
```

在这个版本中，考虑到再条锁并发插入存在死锁（间隙锁争抢）情况，引入中央锁概念。

基本方式是：

1. 根据sql创建好数据库
2. 创建一条记录Flock_name="center_lock"的记录。
3. 在对其他锁(如Flock_name="sale_invite_lock")进行操作的时候，先对"center_lock"记录select for update
4. "sale_invite_lock"记录自己的增删改查。

考虑到不同公司引入的数据库操作包不同，因此提供伪代码，以便于理解 伪代码

java 复制代码

```
// 开启事务
@Transactional
public boolean getLock(String key){
       // 获取中央锁
       select * from tbl where Flock_name="center_lock"

       // 查询key相关记录
       select for update
       if （记录存在）{
             update
       }else {
             insert
    }
}
```

java 复制代码

```
    /**
     * 初始化记录,如果有记录update,如果没有记录insert
     */
    private LockRecord initLockRecord(String key){
        // 查询记录是否存在
        LockRecord lockRecord = lockMapper.queryRecord(key);
        if (null == lockRecord) {
            // 记录不存在，创建
            lockRecord = new LockRecord();
            lockRecord.setLockName(key);
            lockRecord.setCount(0);
            lockRecord.setDesc("");
            lockRecord.setDeadline(new Date(0));
```

```
            lockRecord.setStatus(1);
            lockMapper.insertRecord(lockRecord);
        }
        return lockRecord;
    }

    /**
     * 获取锁，代码片段
     */
    @Override
    @Transactional
    public GetLockResponse getLock(GetLockRequest request) {
        // 检测参数
        if(StringUtils.isEmpty(request.lockName)) {
            ResultUtil.throwBusinessException(CreditCardErrorCode.PARAM_INVALID);
        }

        // 兼容参数初始化
        request.expireTime = null==request.expireTime? 31536000: request.expireTime;
        request.desc = Strings.isNullOrEmpty(request.desc)?"":request.desc;
        Long nowTime = new Date().getTime();

        GetLockResponse response = new GetLockResponse();
        response.lock = 0;

        // 获取中央锁，初始化记录
        lockMapper.queryRecordForUpdate("center_lock");
        LockRecord lockRecord = initLockRecord(request.lockName);

        // 未释放锁或未过期，获取失败
        if (lockRecord.getStatus() == 1
                && lockRecord.getDeadline().getTime() > nowTime){
            return response;
        }
```

```
            // 获取锁
            Date deadline = new Date(nowTime + request.expireTime*1000);
            int num = lockMapper.updateRecord(request.lockName, deadline, 0, request.desc, 1);
            response.lock = 1;
            return response;
        }
```

到此，该方案，能够满足我的分布式锁的需求。

但是该方案，有一个比较致命的问题，就是所有记录共享一个锁，并发并不高。

经过测试，开启50*100个线程并发修改，5次耗时平均为8秒。

实现第三版

由于方案二，存在共享同一把中央锁，并发不高的请求。参考concurrentHashMap实现原理，引入分段锁概念，降低锁粒度。

基本方式是：

1. 根据sql创建好数据库
2. 创建100条记录Flock_name="center_lock_xx"的记录(xx为00-99)。
3. 在对其他锁(如Flock_name="sale_invite_lock")进行操作的时候，根据crc32算法找到对应的center_lock_02，先对"center_lock_02"记录select for update
4. "sale_invite_lock"记录自己的增删改查。

伪代码如下：

java 复制代码

```
// 开启事务
@Transactional
public boolean getLock(String key){
       // 获取中央锁
       select * from tbl where Flock_name="center_lock"

      // 查询key相关记录
      select for update
      if （记录存在）{
            update
      }else {
            insert
   }
}
```

java 复制代码

```
    /**
     * 获取中央锁Key
     */
    private boolean getCenterLock(String key){
        String prefix = "center_lock_";
        Long hash = SecurityUtil.crc32(key);
        if (null == hash){
            return false;
        }
        //取crc32中的最后两位值
        Integer len = hash.toString().length();
        String slot = hash.toString().substring(len-2);

        String centerLockKey = prefix + slot;
        lockMapper.queryRecordForUpdate(centerLockKey);
        return true;
    }
```

```
 /**
 * 获取锁
 */
@Override
@Transactional
public GetLockResponse getLock(GetLockRequest request) {
    // 检测参数
    if(StringUtils.isEmpty(request.lockName)) {
        ResultUtil.throwBusinessException(CreditCardErrorCode.PARAM_INVALID);
    }

    // 兼容参数初始化
    request.expireTime = null==request.expireTime? 31536000: request.expireTime;
    request.desc = Strings.isNullOrEmpty(request.desc)?"":request.desc;
    Long nowTime = new Date().getTime();

    GetLockResponse response = new GetLockResponse();
    response.lock = 0;

    // 获取中央锁，初始化记录
    getCenterLock(request.lockName);
    LockRecord lockRecord = initLockRecord(request.lockName);

    // 未释放锁或未过期，获取失败
    if (lockRecord.getStatus() == 1
            && lockRecord.getDeadline().getTime() > nowTime){
        return response;
    }

    // 获取锁
    Date deadline = new Date(nowTime + request.expireTime*1000);
    int num = lockMapper.updateRecord(request.lockName, deadline, 0, request.desc, 1);
    response.lock = 1;
```

```
            return response;
        }
```

经过测试，开启50*100个线程并发修改，5次耗时平均为5秒。相较于版本二几乎有一倍的提升。

至此，完成redis/mysql分布式锁、计数器的实现与应用。

最后

根据不同应用场景，做出如下选择：

1. 高并发、不保证数据一致性：redis锁/计数器
2. 低并发、保证数据一致性：mysql锁/计数器
3. 低并发、不保证数据一致性：你随意
4. 高并发。保证数据一致性：redis锁/计数器 + mysql锁/计数器。

表数据和记录：

欢迎关注我的简书博客，一起成长，一起进步。

www.jianshu.com/u/5a327aab7…

评论