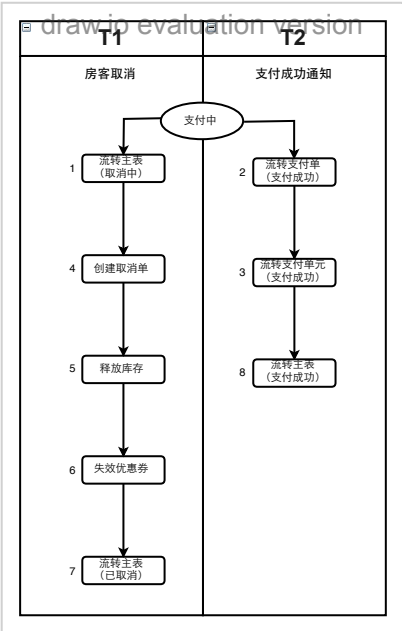
分布式可重入锁

创建： 王丽月，最新修改于： 五月 14, 2017

背景

举栗：



在分布式服务场景下，事务加锁已无法满足业务维度一致性要求。因此，需要一种独立于各服务的控制（存储）中心，保证加锁的唯一性。→ 缓存无疑是最好的选择。此处，我们选择Squirrel作为存储系统。

问答：

1. Q：锁加在哪一层?
   A：加在DB上层，在最后一道防线保证加锁。
2. Q：如果不在最底层加锁，而是在最外层加锁，会有什么问题?
   A：目前，我们许多job是在服务层的，并且这些job仅做服务内部的事情，应该放在服务层。那么，如果最底层的DB没有加锁，就可能出现job和正常的请求同时操作一条数据，那么我们的锁形同虚设。因此，应该在最接近DB的层面保证行级锁。
3. Q：使用方确保不会与job等直接在最底层操作数据，可以只在最外层加锁吗?
   A：可以，实际上，ReenLock是不管你锁加得对不对，它处理你加锁的范围。
4. Q：上层加锁了，下次还可以重复加锁吗?
   A：可以，外层加锁，内层可以对同一key再次加锁，即嵌套锁。只要属于同一请求（这里以traceId为标识），就可以重复加锁，每层执行完成后，层层解锁。这正式起名为ReenLock的原因，ReentrantLock表示可重入锁，但它仅支持线程内重复加锁。而我们的ReenLock支持请求范围内可重复加锁，适应分布式服务需求。
5. Q：嵌套加锁，如果有一层解锁失败了，会不会导致死锁?
   A：解锁是在finally块处理的，不会因为业务代码失败而导致解锁失败。理论上，仅在解锁工具或Squirrel本身出问题的情况下才可能失败。即便在这种情况下，也不会死锁，因为加锁时指定了超时时间（未指定则使用默认值），一旦超过指定时间未加锁，Squirrel将自动删除缓存。

在具体业务场景中，嵌套加锁不可避免，例如，下单环节：

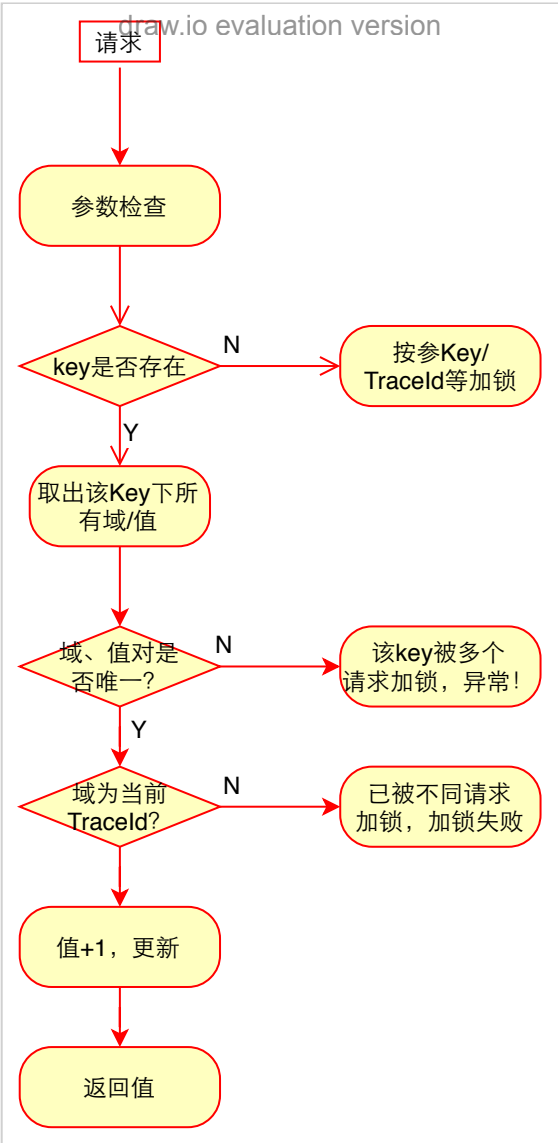
BIZ(lock)：{ (lock)创建订单主表(unlock)→ (lock)创建产品快照(unlock)→ (lock)创建支付单(unlock)→ (lock)创建支付单元(unlock) }

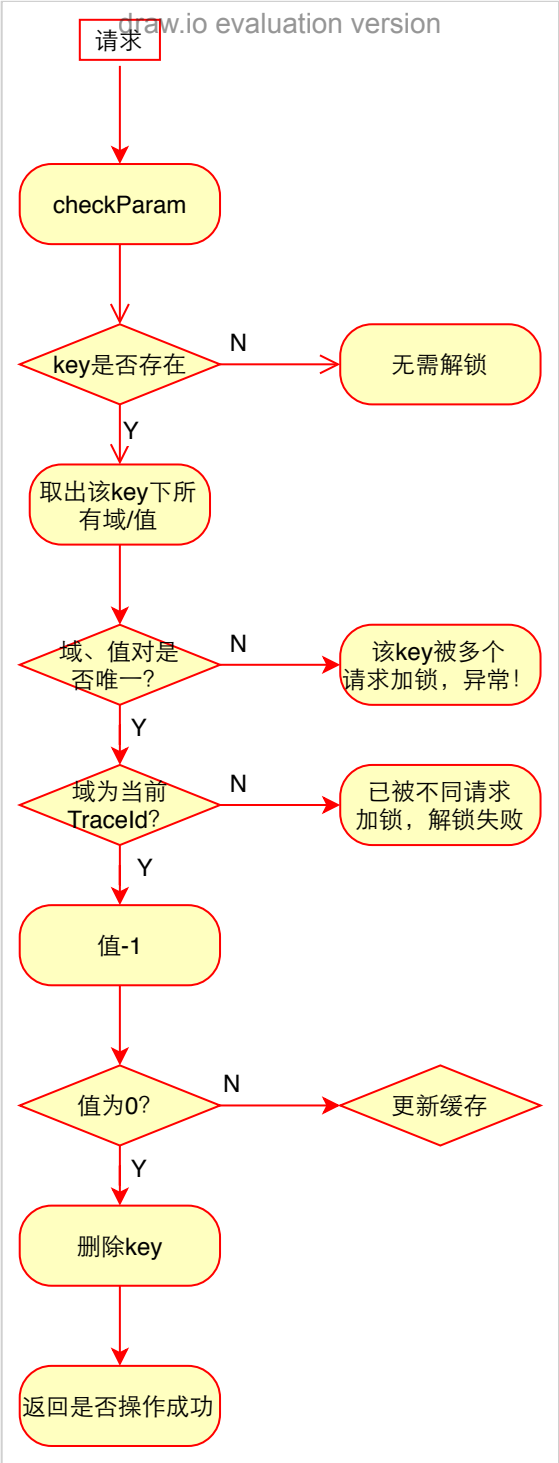
基本原理

目标

1. 同一Key若已被其他请求加锁，则不可重复加锁；2. 同一Key，若已被相同请求（traceId）加锁，则取出加锁次数，+1后更新。

基本流程





加锁逻辑

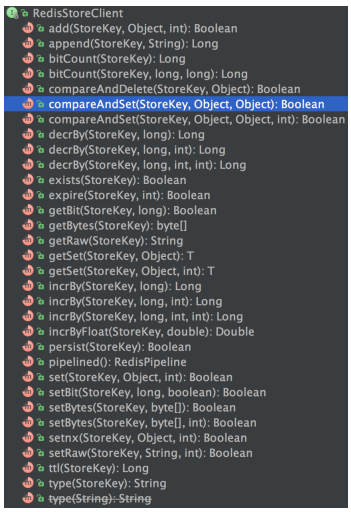
```
Map<String, Integer> map = storeClient.hgetAll(storeKey);
if (map == null || map.size() == 0) {//未曾加锁,直接加锁
    //注意，此处可能存在多个请求同时查出null,并一起写入问题，只要traceId不同，此处的hcompareAndSe
    result = storeClient.hcompareAndSet(storeKey, param.getTraceId(), null, ++value, param.
} else if (map.size() == 1 && map.containsKey(traceId)) {//已加锁,判断是否相同traceId
    value = map.get(traceId);
    result = storeClient.hcompareAndSet(storeKey, param.getTraceId(), value, ++value, param
}
```

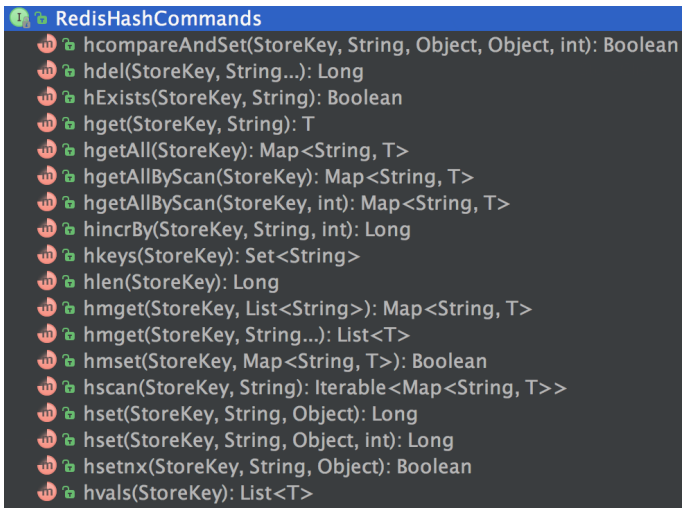
问题：从缓存取数 → 更新缓存，期间操作并非原子的，就可能存在读数与写数之间，缓存已被其他请求更新，由此产生多个traceId同时加了value为1的锁，不符合互斥要求。

那么如何保证读写的原子性呢？如果有人要说用"synchronized"呀。哦，别忘了，这是在分布式服务场景下，加锁发生在不同服务器。理想情况下，我们希望Squirrel能够提供一个加锁操作，由它保证原子性前提下，判断Key是否存在，如果不存在，则设置Key+Field+Value。否则，如果Key存在，但Field不同，那么失败。如果Key+Field存在，则Value++。然而，Squirrel并不能提供如此业务化的需求。因此，在不能够保证读写原子性的情况下，我们采用一种类似乐观锁的机制，先查询，根据查询结果做逻辑判断，但是更新时，一定要确保当前storeKey与查询结果一致。

```
Map<String, Integer> map = storeClient.hgetAll(storeKey);
if (map == null || map.size() == 0) {//未曾加锁,直接加锁
    map = new HashMap<>();
    map.put(param.getTraceId(), ++value);
    result = storeClient.setnx(storeKey, map, param.getExpireTime());
} else if (map.size() == 1 && map.containsKey(traceId)) {//已加锁,判断是否相同traceId
    value = map.get(traceId);
    result = storeClient.hcompareAndSet(storeKey, param.getTraceId(), value, ++value, param
}
```

然而，不幸的是，Squirrel支持的Value类型，要么是Object，使用StoreClient、RedisStoreClient接口相应接口；要么是hash表，使用RedisHashCommands相应接口。对同一storeKey，不支持storeClient.setnx和storeClient.hcompareAndSet同时使用。





最后，因为无法解决初次设置『Key + Field』时的校验，我们只好放弃哈希表的存储方式。

```
Map<String, Integer> map = storeClient.get(storeKey);
if (map == null || map.size() == 0) {//未曾加锁,直接加锁
    map = new HashMap<>();
    map.put(param.getTraceId(), ++value);
    result = storeClient.setnx(storeKey, map, param.getExpireTime());
} else if (map.size() == 1 && map.containsKey(traceId)) {//已加锁,判断是否相同traceId
    value = map.get(traceId);
    Map<String, Integer> newMap = new HashMap<>();
    newMap.put(traceId, ++value);
    result = storeClient.compareAndSet(storeKey, map, newMap, param.getExpireTime());
}
```

质疑

1. 使用Squirrel缓存引擎，是否会存在主从不同步问题？即key已被traceA加锁，而traceB检查是否加锁时，读到的依然是无锁。
   Squirrel是基于redis-cluster的kv存储，集群采用写master读slave，确实存在主从同步问题。对于分布式锁，每次都必须是最实时的，因此，这里可以通过指定com.dianping.squirrel.client.impl.redis.spring.RedisClientBeanFactory#routerType为"master-only"，从主库读。
2. 每次加锁后，过程如出现意外，例如服务器FullGC，导致该锁超时而被释放？并且第二次请求进入，并加锁成功？是否会导致互斥原则被打破？
   一，服务器停顿确实可能导致缓存中锁因超期而释放；并且，第二次加锁也会成功。此时，须得从释放锁处做逻辑。
   A：traceA加锁超时，但未存在其他trace对该key加锁的情况，释放锁发现get(storekey) == null，此时，无需释放锁，直接返回；
   B：traceA加锁超时，且存在traceB对key加锁，则traceA释放锁失败，即traceA的锁可能已被干扰，traceA对应的事务必须回滚。traceB释放锁，正常。

```
if (map == null || map.size() == 0) { //无需解锁,直接返回
    return result;
} else if (map.size() != 1 || !map.containsKey(traceId) || !CompareUtils
        .equals(map.get(traceId), param.getOldValue())) {
    //非唯一锁或非当前traceId锁
    logger.info("reentrantUnlock param:{} has been lock map:{}", param, map);
    throw new APIRuntimeException(IResponseStatusMsg.APIEnum.SYNC_OP_ERROR, "解锁失败")
}
```

使用说明

1. 添加pom依赖

```
<dependency>
    <groupId>com.sankuai.ia</groupId>
    <artifactId>phx-distributed-lock</artifactId>
    <version>1.0.0-SNAPSHOT</version>
</dependency>
```

2. 添加Squirrel集群配置

```
//配置lock缓存使用的集群
phx.lock.cache.cluster.name=redis-hotel-phx_dev
```

3. 添加包扫描路径

```
@SpringBootApplication(scanBasePackages = { "com.sankuai.ia.lock" })
```

或

```
<context:component-scan base-package="com.sankuai.ia.lock"/>
```

4. 调用

方法一：使用@ReenLock或@BatchReenLock（批量加锁）注解

```
@ReenLock(fieldKey = "#orderId", category="phx-order-lock")
public void transferOrder(Long orderId, Integer orderStatus, Long userId) {
    orderService.transfer(orderId, orderStatus, userId);
}

@BatchReenLock(fieldKey = "#orderIds", category="phx-order-lock")
```

```
public void batchTransferOrder(List<Long> orderIds, Integer orderStatus, Long userId) {
    orderService.batchTransfer(orderIds, orderStatus, userId);
}
```

其中，属性"fieldKey"表示使用方法签名中的orderId作为加锁的key，它使用SpEL表达式语法。category则表示加锁的业务分类，以区分不同的业务方。

方法二：直接调用加锁、解锁方法。使用改方式需要记得unlock要放在finally块中，以避免异常情况下没有释放锁。

```
SquirrelLock lock = SquirrelLock.getInstance();
ReentrantLockParam lockParam = new ReentrantLockParam();
lockParam.setCategory(ReenLockConsts.DEFAULT_CATEGORY);
Integer traceNum = (int) (Math.floor((Math.random() * 5) + 1));
String traceId = String.valueOf(traceNum);
lockParam.setTraceId(traceId); //traceId必填
lockParam.setKey("test"); //key必填
int value = lock.reentrantLock(lockParam); //加锁
try {
    logger.info("do something");
} finally {
    ReentrantUnlockParam unlockParam = new ReentrantUnlockParam();
    unlockParam.setCategory(ReenLockConsts.DEFAULT_CATEGORY);
    unlockParam.setKey("test"); //key必填
    unlockParam.setTraceId(traceId); //traceId必填
    unlockParam.setOldValue(value); //加锁返回值必填
    lock.reentrantUnlock(unlockParam); //解锁
}
```

方法三：使用lambda表示式，将待加锁的语句块函数作为参数传递给ReenLockService，有它完成操作前的加锁和操作后的解锁，避免放在客户端代码，遗漏。

```
@Autowired
private ReenLockService reenLockService;
@Test
public void test() {
    ReentrantLockParam param = new ReentrantLockParam();
    param.setKey("1000000");
    param.setTraceId("1");
    param.setCategory(ReenLockConsts.DEFAULT_CATEGORY);
    //支持仅入参，无返回值的Consumer函数
    Consumer<String> consumer = (String arg) -> System.out.println(arg);
    reenLockService.processWithReenLock(param, "test", consumer);
    //支持既有参数又有返回值的Function函数
    Function<Object[], String> function = (Object[] args) -> {
        System.out.println(args);
        return "success";
    };
    String result =  reenLockService.processWithReenLock(param, new Object[]{param}, functi
    assert result.equals("success");
}
```

每次嵌套加锁，更新锁的有效时长，可能在异常的情况，死锁的时长延长。因此，有人提议，仅首次加锁设置超时时间，即最后层锁需要根据整个操作的所需要的最长时间设置expireTime。

暂不考虑

1. 排队等待加锁
2. 优先级锁
3. 读锁

无标签