Redis实现分布式锁的深入探究

转载

公众号:方志朋 于 2020-03-12 19:55:00 发布 25 收藏
文章标签： 分布式 数据库 redis java 编程语言

点击上方“方志朋”，选择“设为星标”

回复”666“获取新整理的面试文章

一、分布式锁简介

锁 是一种用来解决多个执行线程 **访问共享资源** 错误或数据不一致问题的工具。

如果 *把一台服务器比作一个房子*，那么 *线程就好比里面的住户*，当他们想要共同访问一个共享资源，例如厕所的时候，如果厕所门上没有锁...更甚者厕所没装门...这是会出原则性的问题的..

装上了锁，大家用起来就安心多了，本质也就是 **同一时间只允许一个住户使用**。

而随着互联网世界的发展，单体应用已经越来越无法满足复杂互联网的高并发需求，转而慢慢朝着分布式方向发展，慢慢进化成了 **更大一些的住户**。所以同样，我们需要引入分布式锁来解决分布式应用之间访问共享资源的并发问题。

为何需要分布式锁

一般情况下，我们使用分布式锁主要有两个场景：

> **避免不同节点重复相同的工作**：比如用户执行了某个操作有可能不同节点会发送多封邮件；
>
> **避免破坏数据的正确性**：如果两个节点在同一条数据上同时进行操作，可能会造成数据错误或不一致的情况出现；

Java 中实现的常见方式

上面我们用简单的比喻说明了锁的本质：**同一时间只允许一个用户操作**。所以理论上，能够满足这个需求的工具我们都能够使用 *(就是其他应用能帮我们加锁的)*：

> **基于 MySQL 中的锁**：MySQL 本身有自带的悲观锁 `for update` 关键字，也可以自己实现悲观/乐观锁来达到目的；
>
> **基于 Zookeeper 有序节点**：Zookeeper 允许临时创建有序的子节点，这样客户端获取节点列表时，就能够当前子节点列表中的序号判断是否能够获得锁；

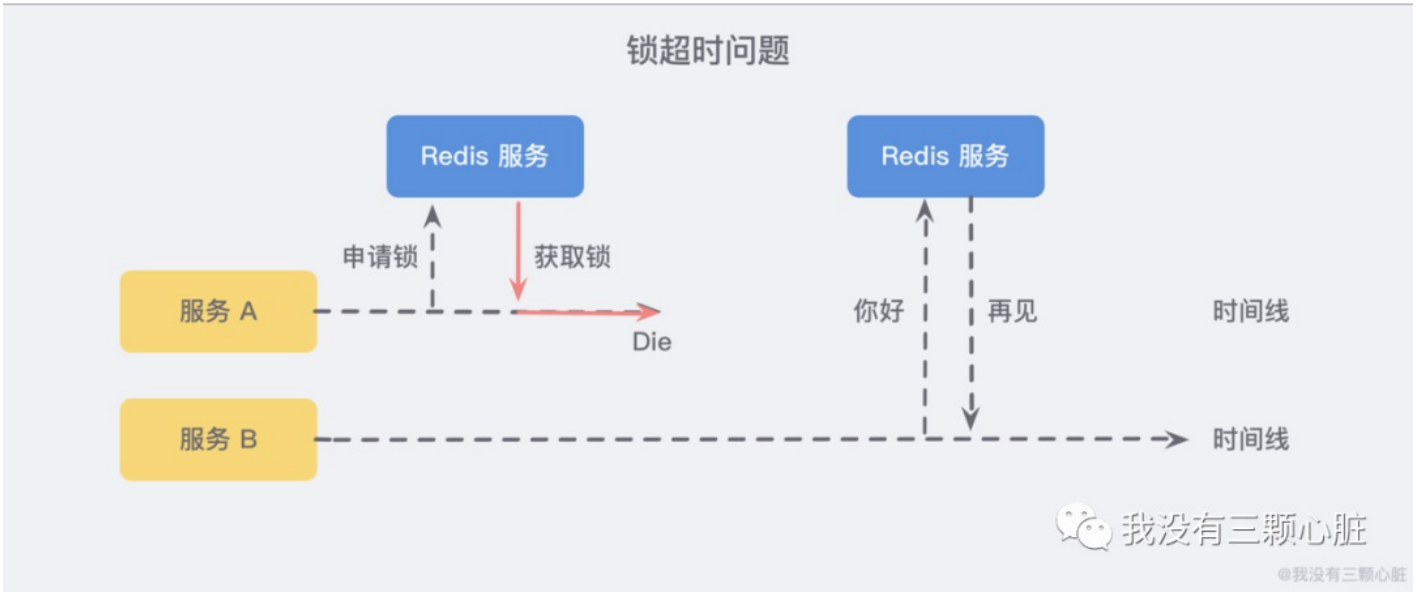
> **基于 Redis 的单线程**：由于 Redis 是单线程，所以命令会以串行的方式执行，并且本身提供了像 `SETNX(set if not exists)` 这样的指令，本身具有互斥性；

每个方案都有各自的优缺点，例如 MySQL 虽然直观理解容易，但是实现起来却需要额外考虑 **锁超时**、**加事务** 等，并且性能局限于数据库，诸如此类我们在此不作讨论，重点关注 Redis。

Redis 分布式锁的问题

1）锁超时

假设现在我们有两台平行的服务 A B，其中 A 服务在 **获取锁之后** 由于未知神秘力量突然 **挂了**，那么 B 服务就永远无法获取到锁了：



所以我们需要额外设置一个超时时间，来保证服务的可用性。

但是另一个问题随即而来：**如果在加锁和释放锁之间的逻辑执行得太长，以至于超出了锁的超时限制**，也会出现问题。因为这时候第一个线程持有锁过期了，而临界区的逻辑还没有执行完，与此同时第二个线程就提前拥有了这把锁，导致临界区的代码不能得到严格的串行执行。

为了避免这个问题，**Redis 分布式锁不要用于较长时间的任务**。如果真的偶尔出现了问题，造成的数据小错乱可能就需要人工的干预。

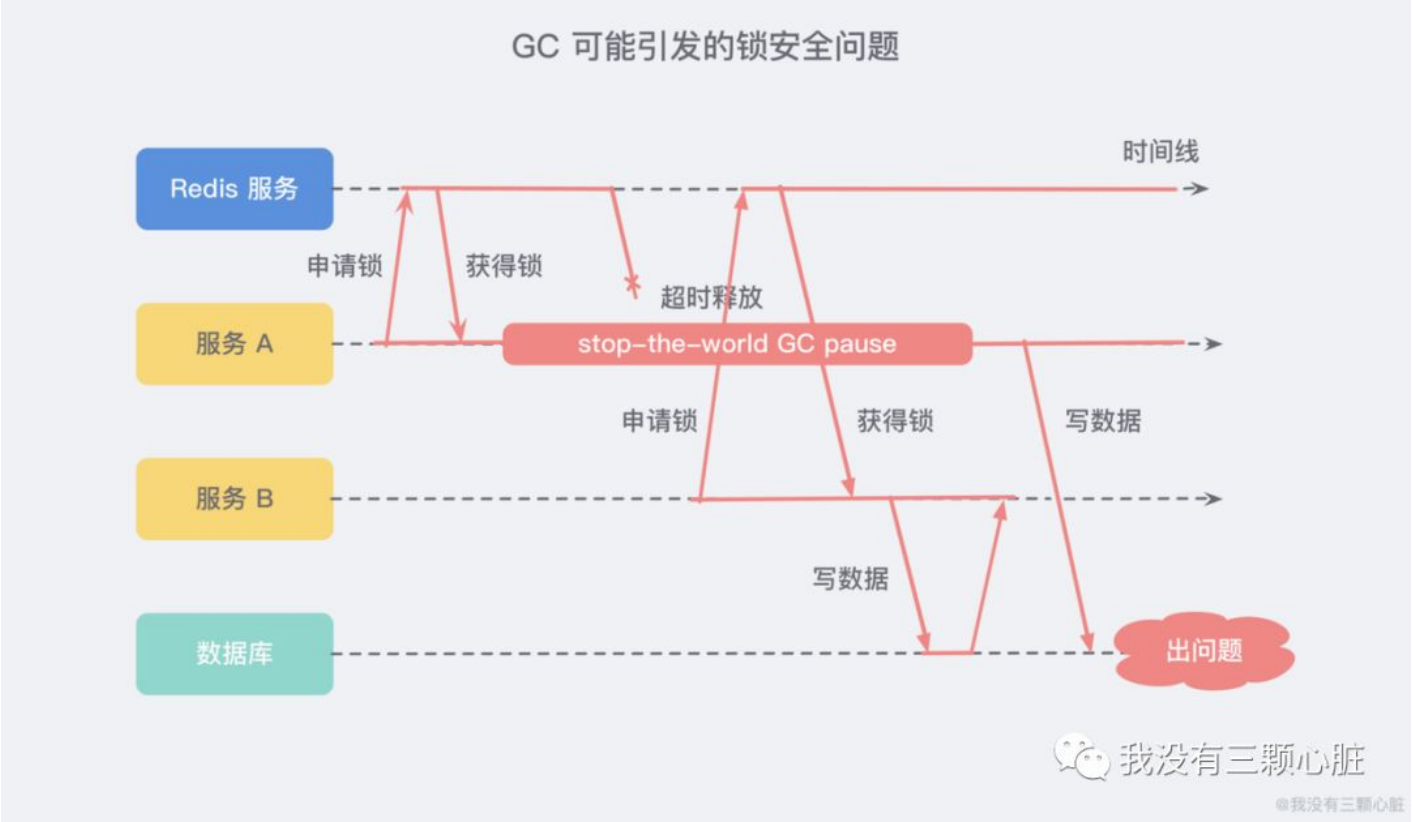
有一个稍微安全一点的方案是 **将锁的 `value` 值设置为一个随机数**，释放锁时先匹配随机数是否一致，然后再删除 key，这是为了 **确保当前线程占有的锁不会被其他线程释放**，除非这个锁是因为过期了而被服务器自动释放的。

但是匹配 `value` 和删除 `key` 在 Redis 中并不是一个原子性的操作，也没有类似保证原子性的指令，所以可能需要使用像 Lua 这样的脚本来处理了，因为 Lua 脚本可以 **保证多个指令的原子性执行**。

延伸的讨论：GC 可能引发的安全问题

Martin Kleppmann 曾与 Redis 之父 Antirez 就 Redis 实现分布式锁的安全性问题进行过深入的讨论，其中有一个问题就涉及到 **GC**。

熟悉 Java 的同学肯定对 GC 不陌生，在 GC 的时候会发生 **STW(Stop-The-World)**，这本身是为了保障垃圾回收器的正常执行，但可能会引发如下的问题：



服务 A 获取了锁并设置了超时时间，但是服务 A 出现了 STW 且时间较长，导致了分布式锁进行了超时释放，在这个期间服务 B 获取到了锁，待服务 A STW 结束之后又恢复了锁，这就导致了 **服务 A 和服务 B 同时获取到了锁**，这个时候分布式锁就不安全了。

不仅仅局限于 Redis，Zookeeper 和 MySQL 有同样的问题。

想吃更多瓜的童鞋，可以访问下列网站看看 Redis 之父 Antirez 怎么说：http://antirez.com/news/101

2）单点/多点问题

如果 Redis 采用单机部署模式，那就意味着当 Redis 故障了，就会导致整个服务不可用。

而如果采用主从模式部署，我们想象一个这样的场景：*服务 A* 申请到一把锁之后，如果作为主机的 Redis 宕机了，那么 *服务 B* 在申请锁的时候就会从从机那里获取到这把锁，为了解决这个问题，Redis 作者提出了一种 **RedLock 红锁** 的算法 *(Redission 同 Jedis)*：

```
// 三个 Redis 集群
RLock lock1 = redissionInstance1.getLock("lock1");
RLock lock2 = redissionInstance2.getLock("lock2");
RLock lock3 = redissionInstance3.getLock("lock3");

RedissionRedLock lock = new RedissionLock(lock1, lock2, lock2);
lock.lock();
// do something....
lock.unlock();
```

二、Redis 分布式锁的实现

分布式锁类似于 "占坑"，而 `SETNX(SET if Not eXists)` 指令就是这样的一个操作，只允许被一个客户端占有，我们来看看 **源码(t_string.c/setGenericCommand)** 吧：

```
// SET/ SETEX/ SETTEX/ SETNX 最底层实现
void setGenericCommand(client *c, int flags, robj *key, robj *val, robj *expire, int unit, robj *ok_reply,
    long long milliseconds = 0; /* initialized to avoid any harmness warning */

    // 如果定义了 key 的过期时间则保存到上面定义的变量中
    // 如果过期时间设置错误则返回错误信息
    if (expire) {
        if (getLongLongFromObjectOrReply(c, expire, &milliseconds, NULL) != C_OK)
            return;
        if (milliseconds <= 0) {
            addReplyErrorFormat(c,"invalid expire time in %s",c->cmd->name);
            return;
        }
        if (unit == UNIT_SECONDS) milliseconds *= 1000;
    }

    // lookupKeyWrite 函数是为执行写操作而取出 key 的值对象
    // 这里的判断条件是：
    // 1.如果设置了 NX(不存在)，并且在数据库中找到了 key 值
    // 2.或者设置了 XX(存在)，并且在数据库中没有找到该 key
    // => 那么回复 abort_reply 给客户端
    if ((flags & OBJ_SET_NX && lookupKeyWrite(c->db,key) != NULL) ||
        (flags & OBJ_SET_XX && lookupKeyWrite(c->db,key) == NULL))
    {
        addReply(c, abort_reply ? abort_reply : shared.null[c->resp]);
        return;
    }

    // 在当前的数据库中设置键为 key 值为 value 的数据
    genericSetKey(c->db,key,val,flags & OBJ_SET_KEEPTTL);
    // 服务器每修改一个 key 后都会修改 dirty 值
    server.dirty++;
    if (expire) setExpire(c,c->db,key,mstime()+milliseconds);
    notifyKeyspaceEvent(NOTIFY_STRING,"set",key,c->db->id);
    if (expire) notifyKeyspaceEvent(NOTIFY_GENERIC,
        "expire",key,c->db->id);
    addReply(c, ok_reply ? ok_reply : shared.ok);
}
```

就像上面介绍的那样，其实在之前版本的 Redis 中，由于 `SETNX` 和 `EXPIRE` 并不是 **原子指令**，所以在一起执行会出现问题。

也许你会想到使用 Redis 事务来解决，但在这里不行，因为 `EXPIRE` 命令依赖于 `SETNX` 的执行结果，而事务中没有 `if-else` 的分支逻辑，如果 `SETNX` 没有抢到锁，`EXPIRE` 就不应该执行。

为了解决这个疑难问题，Redis 开源社区涌现了许多分布式锁的 library，为了治理这个乱象，后来在 Redis 2.8 的版本中，加入了 `SET` 指令的扩展参数，使得 `SETNX` 可以和 `EXPIRE` 指令一起执行了：

```
> SET lock:test true ex 5 nx
OK
... do something critical ...
> del lock:test
```

你只需要符合 `SET key value [EX seconds | PX milliseconds] [NX | XX] [KEEPTTL]` 这样的格式就好了，你也在下方右拐参照官方的文档：

官方文档：https://redis.io/commands/set

另外，官方文档也在 `SETNX` 文档中提到了这样一种思路：**把 SETNX 对应 key 的 value 设置为 <current Unix time + lock timeout + 1>**，这样在其他客户端访问时就能够自己判断是否能够获取下一个 value 为上述格式的锁了。

代码实现

下面用 Jedis 来模拟实现一下，关键代码如下：

```
private static final String LOCK_SUCCESS = "OK";
private static final Long RELEASE_SUCCESS = 1L;
private static final String SET_IF_NOT_EXIST = "NX";
private static final String SET_WITH_EXPIRE_TIME = "PX";

@Override
public String acquire() {
    try {
        // 获取锁的超时时间，超过这个时间则放弃获取锁
        long end = System.currentTimeMillis() + acquireTimeout;
        // 随机生成一个 value
        String requireToken = UUID.randomUUID().toString();
        while (System.currentTimeMillis() < end) {
            String result = jedis
                .set(lockKey, requireToken, SET_IF_NOT_EXIST, SET_WITH_EXPIRE_TIME, expireTime);
            if (LOCK_SUCCESS.equals(result)) {
                return requireToken;
            }
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    } catch (Exception e) {
        log.error("acquire lock due to error", e);
    }

    return null;
}

@Override
public boolean release(String identify) {
    if (identify == null) {
        return false;
    }

    String script = "if redis.call('get', KEYS[1]) == ARGV[1] then return redis.call('del', KEYS[1]) else r
    Object result = new Object();
    try {
        result = jedis.eval(script, Collections.singletonList(lockKey),
            Collections.singletonList(identify));
        if (RELEASE_SUCCESS.equals(result)) {
            log.info("release lock success, requestToken:{}", identify);
            return true;
        }
    } catch (Exception e) {
        log.error("release lock due to error", e);
    } finally {
        if (jedis != null) {
            jedis.close();
        }
    }

    log.info("release lock failed, requestToken:{}, result:{}", identify, result);
    return false;
}
```

引用自下方 *参考资料 3*，其中还有 RedLock 的实现和测试，有兴趣的童鞋可以戳一下

推荐阅读

【官方文档】Distributed locks with Redis - https://redis.io/topics/distlock

Redis【入门】就这一篇! - https://www.wmyskxz.com/2018/05/31/redis-ru-men-jiu-zhe-yi-pian/

Redission - Redis Java Client 源码 - https://github.com/redisson/redisson

手写一个 Jedis 以及 JedisPool - https://juejin.im/post/5e5101c46fb9a07cab3a953a

参考资料

再有人问你分布式锁，这篇文章扔给他 - https://juejin.im/post/5bbb0d8df265da0abd3533a5#heading-0

【官方文档】Distributed locks with Redis - https://redis.io/topics/distlock

【分布式缓存系列】Redis实现分布式锁的正确姿势 - https://www.cnblogs.com/zhili/p/redisdistributelock.html

Redis源码剖析和注释（九）--- 字符串命令的实现(t_string) - https://blog.csdn.net/men_wen/article/details/70325566

《Redis 深度历险》 - 钱文品/ 著