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。

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。

二、为何需要分布式锁

一般情况下，使用分布式锁主要有两个场景：

1. **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；
2. **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；

三、Java中实现的常见方式

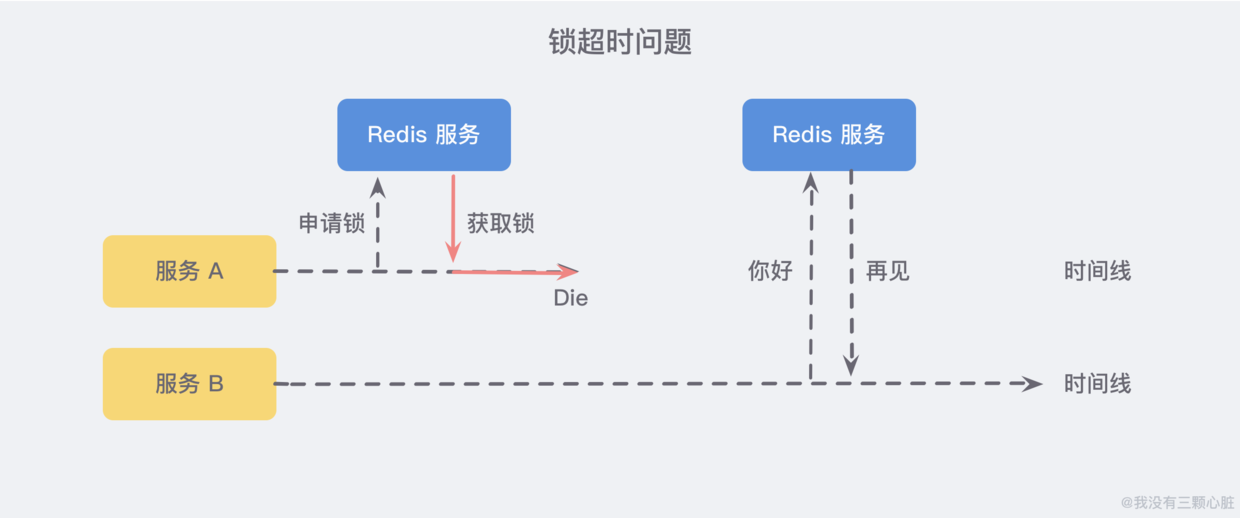
锁的本质：同一时间只允许一个用户操作。所以理论上，能够满足这个需求的工具我们都能够使用：

1. 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；
2. 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，就能够根据当前子节点列表中的序号判断是否能够获得锁。
3. 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；

四、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位置神秘力量突然挂了，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。

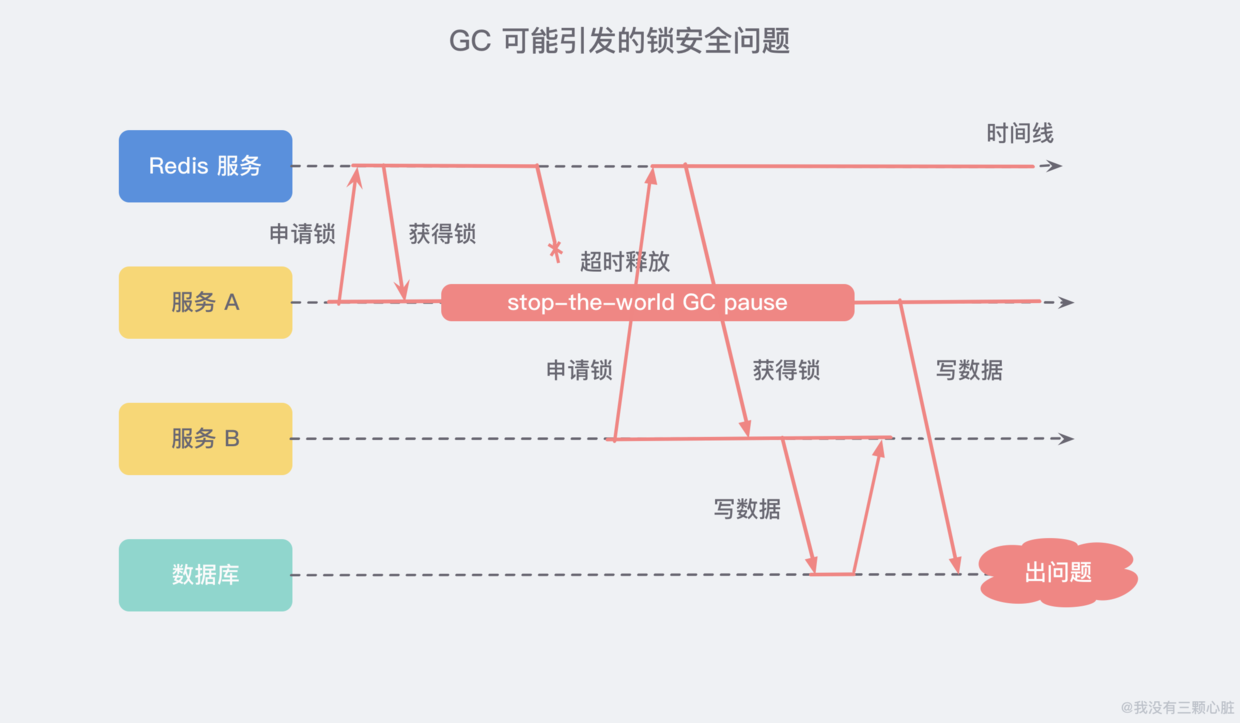
但是另一个问题随即而来：如果在加锁和释放锁之间的逻辑执行得太长，以至于超出了锁的超时限制，也会出现问题。因为这时候第一个线程持有锁过期了，而临界区的逻辑还没有执行完，与此同时第二个线程就提前拥有了这把锁，导致临界区的代码不能得到严格的串行执行。

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。

#### 2、GC可能引发的安全问题

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。

不仅局限于Redis，Zookeeper和MySQL有同样的问题。

#### 3、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();

五、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，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, robj \*abort\_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，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

只需要符合SET key value [EX seconds | PX milliseconds] [NX | XX] [KEEPTTL]这样的格式就好了。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