分布式锁的几种实现方式

题目标签

学习时长：20分钟

题目难度：中等

知识点标签：分布式锁 数据库 缓存 Zookeeper

题目描述

分布式锁的几种实现方式

1. 面试题分析

1. 根据题目要求我们可以知道：

    1、分布式锁的几种实现方式

    2、基于数据库实现分布式锁

3、基于缓存实现分布式锁

4、基于Zookeeper实现分布式锁

5、三种方案的比较

2. 容易被忽略的坑

- 分析片面
- 没有深入

01分布式锁的几种实现方式

目前几乎很多大型网站及应用都是分布式部署的，分布式场景中的数据一致性问题一直是一个比较重要的话题。

分布式的CAP理论告诉我们，任何一个分布式系统都无法同时满足一致性（Consistency）、可用性（Availability）和分区容错性（Partition tolerance），最多只能同时满足两项。

所以，很多系统在设计之初就要对这三者做出取舍。在互联网领域的绝大多数的场景中，都需要牺牲强一致性来换取系统的高可用性，系统往往只需要保证“最终一致性”，只要这个最终时间是在用户可以接受的范围内即可。

在很多场景中，我们为了保证数据的最终一致性，需要很多的技术方案来支持，比如分布式事务、分布式锁等。有的时候，我们需要保证一个方法在同一时间内只能被同一个线程执行。在单机环境中，Java中其实提供了很多并发处理相关的API，但是这些API在分布式场景中就无能为力了。

也就是说单纯的Java Api并不能提供分布式锁的能力。所以针对分布式锁的实现目前有多种方案。

针对分布式锁的实现，目前比较常用的有以下几种方案：

1.数据库实现

2.基于缓存（redis，memcached等）实现

3.Zookeeper实现分布式锁

在分析这几种实现方案之前我们先来想一下，我们需要的分布式锁应该是怎么样的？（这里以方法锁为例，资源锁同理）

> *1）可以保证在分布式部署的应用集群中，同一个方法在同一时间只能被一台机器上的一个线程执行。*
>
> *2）这把锁要是一把可重入锁（避免死锁）*
>
> *3）这把锁最好是一把阻塞锁（根据业务需求考虑要不要这条）*
>
> *4）有高可用的获取锁和释放锁功能*

> *5）获取锁和释放锁的性能要好*

02基于数据库实现分布式锁

1.基于数据库表

要实现分布式锁，最简单的方式可能就是直接创建一张锁表，然后通过操作该表中的数据来实现了。

当我们要锁住某个方法或资源时，我们就在该表中增加一条记录，想要释放锁的时候就删除这条记录。

创建这样一张数据库表：

```
CREATE TABLE `methodLock` (
  `id` int(11) NOT NULL AUTO_INCREMENT COMMENT '主键',
  `method_name` varchar(64) NOT NULL DEFAULT '' COMMENT '锁定的方法名',
  `desc` varchar(1024) NOT NULL DEFAULT '备注信息',
  `update_time` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP COMMENT '保存数据时间，自动生
成',
  PRIMARY KEY (`id`),
  UNIQUE KEY `uidx_method_name` (`method_name `) USING BTREE
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='锁定中的方法';
```

当我们想要锁住某个方法时，执行以下SQL：

```
insert into methodLock(method_name,desc) values (‘method_name’,‘desc’)
```

因为我们对method_name做了唯一性约束，这里如果有多个请求同时提交到数据库的话，数据库会保证只有一个操作可以成功，那么我们就可以认为操作成功的那个线程获得了该方法的锁，可以执行方法体内容。

当方法执行完毕之后，想要释放锁的话，需要执行以下Sql:

```
delete from methodLock where method_name ='method_name'
```

上面这种简单的实现有以下几个问题：

> *1、这把锁强依赖数据库的可用性，数据库是一个单点，一旦数据库挂掉，会导致业务系统不可用。*
>
> *2、这把锁没有失效时间，一旦解锁操作失败，就会导致锁记录一直在数据库中，其他线程无法再获得到锁。*
>
> *3、这把锁只能是非阻塞的，因为数据的insert操作，一旦插入失败就会直接报错。没有获得锁的线程并不会进入排队队列，要想再次获得锁就要再次触发获得锁操作。*
>
> *4、这把锁是非重入的，同一个线程在没有释放锁之前无法再次获得该锁。因为数据中数据已经存在了。*

当然，我们也可以有其他方式解决上面的问题。

- 数据库是单点？搞两个数据库，数据之前双向同步。一旦挂掉快速切换到备库上。
- 没有失效时间？只要做一个定时任务，每隔一定时间把数据库中的超时数据清理一遍。
- 非阻塞的？搞一个while循环，直到insert成功再返回成功。

- 非重入的？在数据库表中加个字段，记录当前获得锁的机器的主机信息和线程信息，那么下次再获取锁的时候先查询数据库，如果当前机器的主机信息和线程信息在数据库可以查到的话，直接把锁分配给他就可以了。

2.基于数据库排他锁

除了可以通过增删操作数据表中的记录以外，其实还可以借助数据中自带的锁来实现分布式的锁。

我们还用刚刚创建的那张数据库表。可以通过数据库的排他锁来实现分布式锁。 基于MySql的InnoDB引擎，可以使用以下方法来实现加锁操作：

```
public boolean lock(){
    connection.setAutoCommit(false)
    while(true){
        try{
            result = select * from methodLock where method_name=xxx for update;
            if(result==null){
                return true;
            }
        }catch(Exception e){

        }
        sleep(1000);
    }
    return false;
}
```

在查询语句后面增加for update，数据库会在查询过程中给数据库表增加排他锁（这里再多提一句，InnoDB引擎在加锁的时候，只有通过索引进行检索的时候才会使用行级锁，否则会使用表级锁。这里我们希望使用行级锁，就要给method_name添加索引，值得注意的是，这个索引一定要创建成唯一索引，否则会出现多个重载方法之间无法同时被访问

的问题。重载方法的话建议把参数类型也加上。）。当某条记录被加上排他锁之后，其他线程无法再在该行记录上增加排他锁。

我们可以认为获得排它锁的线程即可获得分布式锁，当获取到锁之后，可以执行方法的业务逻辑，执行完方法之后，再通过以下方法解锁：

```
public void unlock(){
    connection.commit();
}
```

通过connection.commit()操作来释放锁。

这种方法可以有效的解决上面提到的无法释放锁和阻塞锁的问题。

- 阻塞锁？ for update语句会在执行成功后立即返回，在执行失败时一直处于阻塞状态，直到成功。
- 锁定之后服务宕机，无法释放？使用这种方式，服务宕机之后数据库会自己把锁释放掉。

但是还是无法直接解决数据库单点和可重入问题。

这里还可能存在另外一个问题，虽然我们对method_name 使用了唯一索引，并且显示使用for update来使用行级锁。但是，MySql会对查询进行优化，即便在条件中使用了索引字段，但是否使用索引来检索数据是由 MySQL 通过判断不同执行计划的代价来决定的，如果 MySQL 认为全表扫效率更高，比如对一些很小的表，它就不会使用索引，这种情况下

InnoDB 将使用表锁，而不是行锁。如果发生这种情况就悲剧了。。。

还有一个问题，就是我们要使用排他锁来进行分布式锁的 lock，那么一个排他锁长时间不提交，就会占用数据库连接。一旦类似的连接变得多了，就可能把数据库连接池撑爆

数据库实现分布式锁总结

总结一下使用数据库来实现分布式锁的方式，这两种方式都是依赖数据库的一张表，一种是通过表中的记录的存在情况确定当前是否有锁存在，另外一种是通过数据库的排他锁来实现分布式锁。

数据库实现分布式锁的优点

- 直接借助数据库，容易理解。

数据库实现分布式锁的缺点

- 会有各种各样的问题，在解决问题的过程中会使整个方案变得越来越复杂。
- 操作数据库需要一定的开销，性能问题需要考虑。
- 使用数据库的行级锁并不一定靠谱，尤其是当我们的锁表并不大的时候。

03基于缓存实现分布式锁

相比较于基于数据库实现分布式锁的方案来说，基于缓存来实现在性能方面会表现的更好一点。而且很多缓存是可以集群部署的，可以解决单点问题。

目前有很多成熟的缓存产品，包括Redis，memcached以及我们公司内部的Tair。

这里以Tair为例来分析下使用缓存实现分布式锁的方案。关于Redis和memcached在网络上有很多相关的文章，并且也有一些成熟的框架及算法可以直接使用。

基于Tair的实现分布式锁其实和Redis类似，其中主要的实现方式是使用TairManager.put方法来实现。

```
public boolean trylock(String key) {
    ResultCode code = ldbTairManager.put(NAMESPACE, key, "This is a Lock.", 2, 0);
    if (ResultCode.SUCCESS.equals(code))
        return true;
    else
        return false;
}
public boolean unlock(String key) {
    ldbTairManager.invalid(NAMESPACE, key);
}
```

以上实现方式同样存在几个问题：

> *1、这把锁没有失效时间，一旦解锁操作失败，就会导致锁记录一直在tair中，其他线程无法再获得到锁。*
>
> *2、这把锁只能是非阻塞的，无论成功还是失败都直接返回。*

> *3、这把锁是非重入的，一个线程获得锁之后，在释放锁之前，无法再次获得该锁，因为使用到的key在tair中已经存在。无法再执行put操作。*

当然，同样有方式可以解决。

- 没有失效时间？tair的put方法支持传入失效时间，到达时间之后数据会自动删除。
- 非阻塞？while重复执行。
- 非可重入？在一个线程获取到锁之后，把当前主机信息和线程信息保存起来，下次再获取之前先检查自己是不是当前锁的拥有者。

但是，失效时间我设置多长时间为好？如何设置的失效时间太短，方法没等执行完，锁就自动释放了，那么就会产生并发问题。如果设置的时间太长，其他获取锁的线程就可能要平白的多等一段时间。这个问题使用数据库实现分布式锁同样存在

缓存实现分布式锁总结

可以使用缓存来代替数据库来实现分布式锁，这个可以提供更好的性能，同时，很多缓存服务都是集群部署的，可以避免单点问题。并且很多缓存服务都提供了可以用来实现分布式锁的方法，比如Tair的put方法，redis的setnx方法等。并且，这些缓存服务也都提供了对数据的过期自动删除的支持，可以直接设置超时时间来控制锁的释放。

缓存实现分布式锁的优点

- 性能好，实现起来较为方便。

缓存实现分布式锁的缺点

- 通过超时时间来控制锁的失效时间并不是十分的靠谱。

04基于Zookeeper实现分布式锁

基于zookeeper临时有序节点可以实现的分布式锁。

大致思想即为：每个客户端对某个方法加锁时，在zookeeper上的与该方法对应的指定节点的目录下，生成一个唯一的瞬时有序节点。 判断是否获取锁的方式很简单，只需要判断有序节点中序号最小的一个。 当释放锁的时候，只需将这个瞬时节点删除即可。同时，其可以避免服务宕机导致的锁无法释放，而产生的死锁问题。

来看下Zookeeper能不能解决前面提到的问题。

- 锁无法释放？使用Zookeeper可以有效的解决锁无法释放的问题，因为在创建锁的时候，客户端会在ZK中创建一个临时节点，一旦客户端获取到锁之后突然挂掉（Session连接断开），那么这个临时节点就会自动删除掉。其他客户端就可以再次获得锁。
- 非阻塞锁？使用Zookeeper可以实现阻塞的锁，客户端可以通过在ZK中创建顺序节点，并且在节点上绑定监

听器，一旦节点有变化，Zookeeper会通知客户端，客户端可以检查自己创建的节点是不是当前所有节点中序号最小的，如果是，那么自己就获取到锁，便可以执行业务逻辑了。

- 不可重入？使用Zookeeper也可以有效的解决不可重入的问题，客户端在创建节点的时候，把当前客户端的主机信息和线程信息直接写入到节点中，下次想要获取锁的时候和当前最小的节点中的数据比对一下就可以了。如果和自己的信息一样，那么自己直接获取到锁，如果不一样就再创建一个临时的顺序节点，参与排队。
- 单点问题？使用Zookeeper可以有效的解决单点问题，ZK是集群部署的，只要集群中有半数以上的机器存活，就可以对外提供服务。

可以直接使用zookeeper第三方库Curator客户端，这个客户端中封装了一个可重入的锁服务。

```
public boolean tryLock(long timeout, TimeUnit unit) throws InterruptedException {
    try {
        return interProcessMutex.acquire(timeout, unit);
    } catch (Exception e) {
        e.printStackTrace();
    }
    return true;
}
public boolean unlock() {
    try {
        interProcessMutex.release();
    } catch (Throwable e) {
        log.error(e.getMessage(), e);
    } finally {
        executorService.schedule(new Cleaner(client, path), delayTimeForClean, TimeUnit.MILLISECONDS);
    }
    return true;
}
```

Curator提供的InterProcessMutex是分布式锁的实现。acquire方法用户获取锁，release方法用于释放锁。

使用ZK实现的分布式锁好像完全符合了本文开头我们对一个分布式锁的所有期望。但是，其实并不是，Zookeeper实现的分布式锁其实存在一个缺点，那就是性能上可能并没有缓存服务那么高。因为每次在创建锁和释放锁的过程中，都要动态创建、销毁瞬时节点来实现锁功能。ZK中创建和删除节点只能通过Leader服务器来执行，然后将数据同不到所有的Follower机器上。

其实，使用Zookeeper也有可能带来并发问题，只是并不常见而已。考虑这样的情况，由于网络抖动，客户端可ZK集群的session连接断了，那么zk以为客户端挂了，就会删除临时节点，这时候其他客户端就可以获取到分布式锁了。就可能产生并发问题。这个问题不常见是因为zk有重试机制，一旦zk集群检测不到客户端的心跳，就会重试，Curator客户端支持多种重试策略。多次重试之后还不行的话才会删除临时节点。（所以，选择一个合适的重试策略也比较重要，要在锁的粒度和并发之间找一个平衡。）

Zookeeper实现分布式锁总结

Zookeeper实现分布式锁的优点

- 有效的解决单点问题，不可重入问题，非阻塞问题以及锁无法释放的问题。实现起来较为简单。

Zookeeper实现分布式锁的缺点

- 性能上不如使用缓存实现分布式锁。 需要对ZK的原理有所了解。

05三种方案的比较

上面几种方式，哪种方式都无法做到完美。就像CAP一样，在复杂性、可靠性、性能等方面无法同时满足，所以，根据不同的应用场景选择最适合自己的才是王道。

1.从理解的难易程度角度（从低到高）

数据库 > 缓存 > Zookeeper

2.从实现的复杂性角度（从低到高）

Zookeeper >= 缓存 > 数据库

3.从性能角度（从高到低）

缓存 > Zookeeper >= 数据库

4.从可靠性角度（从高到低）

Zookeeper > 缓存 > 数据库

2. 扩展内容

- 分布式Session共享的4类技术方案，与优劣势比较
- 分布式事务的解决方案，以及原理、总结
- 分布式锁的由来、及Redis分布式锁的实现详解