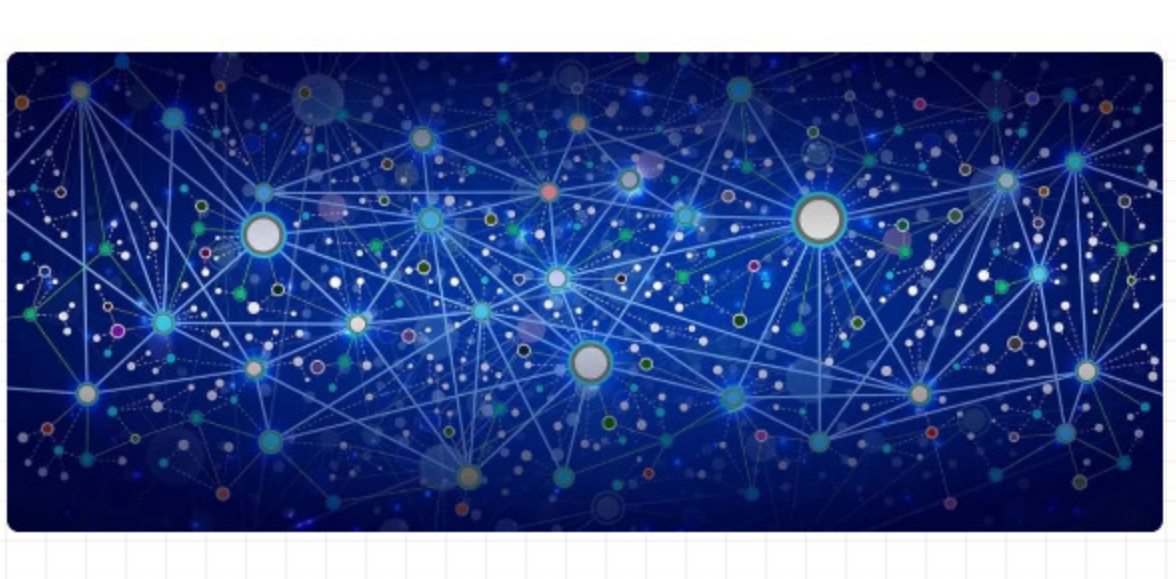
关于分布式锁的面试题都在这里了

程序员面试 5月30日

以下文章来源于转行程序员，作者转行程序员

转行程序员
你要再说我写的文章没用我可就要报警了！！



「我今天班儿都没上，就为了赶紧把这篇文章分布式锁早点写完。我真的不能再贴心了。」

「边喝茶边构思，你们可不要白嫖了！三连来一遍？」

引言

为什么要学习分布式锁？

最简单的理由就是作为一个社招程序员，面试的时候一定被面啦，你看怎么多公众号都翻来覆去的发分布式锁的主题，可见它很重要啦，在高考里这就是送分题，不要怪可惜的。

那应届生也会问吗？这就不一定了，但是，如果你会，面试官肯定会多给你那点分（钱）

第三，分布式锁在稍微有丢丢规模大系统里是必备技能啦。认真看看吧。

分布式锁要解决的问题

分布式锁是一个在分布式环境中的重要原语，它表明不同进程间采用互斥的方式操作共享资源。常见的场景是作为一个sdk被引入到大型项目中，主要解决两类问题：

- 提升效率：加锁是为了避免不必要的重复处理。例如防止幂等任务被多个执行者抢占。此时对锁的正确性要求不高；
- 保证正确性：加锁是为了避免Race Condition导致逻辑错误。例如直接使用分布式锁实现防重，幂等机制。此时如果锁出现错误会引起严重后果，因此对锁的正确性要求高。

Java里的锁：

锁是开发过程中十分常见的工具，你一定不陌生，悲观锁，乐观锁，排它锁，公平锁，非公平锁等等，很多概念，如果你对java里的锁还不了解，可以参考这一篇：不可不说的Java"锁"事（https://tech.meituan.com/2018/11/15/java-lock.html），这一篇写的很全面了，但是对于初学者，知道这些锁的概念，由于缺乏实际工作经验，可能并不了解锁的实际使用场景，Java中可以通过Volatile、Synchronized、ReentrantLock 三个关键字来实现线程的安全，这部分知识在第一轮基础面试里一定会问（要熟练掌握哦）。

在分布式系统中Java这些锁技术是无法同时锁住两台机器上的代码，所以要通过分布式锁来实现，熟练使用分布式锁也是大厂开发必会的技能。

1.面试官：你有遇到需要使用分布式锁的场景吗？

「问题分析：」

这个问题主要作为引子，先要了解什么场景下需要使用分布式锁，分布式锁要解决什么问题，在此前提下有助于你更好的理解分布式锁的实现原理。

使用分布式锁的场景一般需要满足以下场景：

1. 系统是一个分布式系统，java的锁已经锁不住了。
2. 操作共享资源，比如库里唯一的用户数据。
3. 同步访问，即多个进程同时操作共享资源。

「我：」

说一个我在项目中使用分布式锁场景的例子：

消费积分在很多系统里都有，信用卡，电商网站，通过积分换礼品等，这里"**「消费积分」**"这个操作典型的需要使用锁的场景。

「事件A：」 以积分兑换礼品为例来讲，完整的积分消费过程简单分成3步：

A1：用户选中商品，发起兑换提交订单。

A2：系统读取用户剩余积分：判断用户当前积分是否充足。

A3：扣掉用户积分。

「事件B：」 系统给用户发放积分也简单分成3步：

B1：计算用户当天应得积分

B2：读取用户原有积分

B3：在原有积分上增加本次应得积分

那么问题来了，如果用户消费积分和用户累加积分同时发生（同时用户积分进行操作）会怎样？

「假设」：用户在消费积分的同时恰好离线任务在计算积分给用户发放积分（如根据用户当天的消费额），这两件事同时进行，下面的逻辑有点绕，耐心理解。

用户U有1000积分（记录用户积分的数据可以理解为**「共享资源」**），本次兑换要消耗掉999积分。

「不加锁的情况：「事件A程序在执行到第2步读积分时，A:2操作读到的结果是1000分，判断剩余积分够本次兑换，紧接着要执行第3步A:3操作扣积分（1000 - 999 = 1），正常结果应该是用户还是1分。但是这个时候」事件B」也在执行，本次要给用户U发放100积分，两个线程同时进行（**「同步访问」**），不加锁的情况，就会有下面这种可能，A:2 -> B:2 -> A:3 -> B:3 ,在A:3尚未完成前（扣积分，1000 - 999），用户U总积分被事件B的线程读取了，**「最后用户U的总积分变成了1100分，还白白兑换了一个999积分的礼物，这显然不符合预期结果。」**

有人说怎么可能这么巧同时操作用户积分，cpu那么快，只要用户足够多，并发量足够大，墨菲定律迟早生效，出现上述bug只是时间问题，还有可能被黑产行业卡住这个bug疯狂薅羊毛，这个时候作为开发人员要解决这个隐患就必须了解锁的使用。

（写代码是一项严谨的事儿！）

Java本身提供了两种内置的锁的实现，一种是由JVM实现的synchronized 和 JDK 提供的 Lock，以及很多原子操作类都是线程安全的，当你的应用是单机或者说单进程应用时，可以使用这两种锁来实现锁。

但是当下互联网公司的系统几乎都是分布式的，这个时候Java自带的 synchronized 或 Lock 已经无法满足分布式环境下锁的要求了，因为代码会部署在多台机器上，为了解决这个问题，分布式锁应运而生，分布式锁的特点是多进程，多个物理机器上无法共享内存，常见的解决办法是基于内存层的干涉，落地方案就是基于Redis的分布式锁 or ZooKeeper分布式锁。

（我分析的不能更详细了，面试官再不满意？）

2.面试官：那常见的分布式锁有哪些解决方案，你有了解吗？

我：常见的就三种办法吧！

1. Reids的分布式锁，很多大公司会基于Reidis做扩展开发。
2. 基于Zookeeper
3. 基于数据库，比如Mysql。

3.面试官：说说Redis分布式锁实现方法

「问题分析：」

目前分布式锁的实现方式主要有两种，1.基于Redis Cluster模式。2.基于Zookeeper 集群模式。

「优先掌握这两种」，应付面试基本没问题了。

加锁的方式大致有三种，分别是DB分布式锁，Redis分布式锁，Zookepper分布式锁。

「我：」

「1.基于Redis的分布式锁」

「方法一：使用setnx命令加锁」

```java
public static void wrongGetLock1(Jedis jedis, String lockKey, String reques
  // 第一步：加锁
    Long result = jedis.setnx(lockKey, requestId);
    if (result == 1) {
        // 第二步：设置过期时间
        jedis.expire(lockKey, expireTime);
    }

}
```

「代码解释：」

- setnx命令，意思就是 set if not exist，如果lockKey不存在，把key存入Redis，保存成功后如果result返回1，表示设置成功，如果非1，表示失败，别的线程已经设置过了。
- expire()，设置过期时间，防止死锁，假设，如果一个锁set后，一直不删掉，那这个锁相当于一直存在，产生死锁。

（讲到这里，我还要和面试官强调一个"但是"）

思考，我上面的方法哪里与缺陷？继续给面试官解释...

加锁总共分两步，第一步jedis.setnx，第二步jedis.expire设置过期时间，setnx与expire不是一个原子操作，如果程序执行完第一步后异常了，第二步jedis.expire(lockKey, expireTime)没有得到执行，相当于这个锁没有过期时间，有产生死锁的可能。正对这个问题如何改进？

「改进：」

```java
public class RedisLockDemo {

    private static final String SET_IF_NOT_EXIST = "NX";
    private static final String SET_WITH_EXPIRE_TIME = "PX";

    /**
     * 获取分布式锁
     * @param jedis Redis客户端
     * @param lockKey 锁
     * @param requestId 请求标识
     * @param expireTime 超期时间
     * @return 是否获取成功
     */
    public static boolean getLock(Jedis jedis, String lockKey, String reque

    // 两步合二为一，一行代码加锁并设置 + 过期时间。
        if (1 == jedis.set(lockKey, requestId, SET_IF_NOT_EXIST, SET_WITH_E
            return true;//加锁成功
        }
        return false;//加锁失败

    }

}
```

「代码解释：」

将加锁和设置过期时间合二为一，一行代码搞定，原子操作。

(没等面试官开口追问，面试官很满意了)

「面试官：」 那解锁操作呢？

「我：」

释放锁就是删除key

「使用del命令解锁」

```java
public static void unLock(Jedis jedis, String lockKey, String requestId) {

    // 第一步： 使用 requestId 判断加锁与解锁是不是同一个客户端
    if (requestId.equals(jedis.get(lockKey))) {
        // 第二步： 若在此时，这把锁突然不是这个客户端的，则会误解锁
        jedis.del(lockKey);
    }
}
```

「代码解释：」 通过 requestId 判断加锁与解锁是不是同一个客户端和 jedis.del(lockKey) 两步不是原子操作，理论上会出现在执行完第一步if判断操作后锁其实已经过期，并且被其它线程获取，这是时候在执行jedis.del(lockKey)操作，相当于把别人的锁释放了，这是不合理的。当然，这是非常极端的情况，如果unLock方法里第一步和第二步没有其它业务操作，把上面的代码扔到线上，可能也不会真的出现问题，原因第一是业务并发量不高，根本不会暴露这个缺陷，那么问题还不大。

但是写代码是严谨的工作，能完美则必须完美。针对上述代码中的问题，提出改进。

「代码改进：」

```java
public class RedisTool {

    private static final Long RELEASE_SUCCESS = 1L;

    /**
     * 释放分布式锁
     * @param jedis Redis客户端
     * @param lockKey 锁
     * @param requestId 请求标识
     * @return 是否释放成功
     */
    public static boolean releaseDistributedLock(Jedis jedis, String lockKe

        String script = "if redis.call('get', KEYS[1]) == ARGV[1] then retu
        Object result = jedis.eval(script, Collections.singletonList(lockKe

        if (RELEASE_SUCCESS.equals(result)) {
            return true;
        }
        return false;

    }

}
```

「代码解释：」

通过 jedis 客户端的 eval 方法和 script 脚本一行代码搞定，解决方法一中的原子问题。

4.面试官：说说基于 ZooKeeper 的分布式锁实现原理

「我：」

还是积分消费与积分累加的例子：**「事件A」**和**「事件B」**同时需要进行对积分的修改操作，两台机器同时进行，正确的业务逻辑上让一台机器先执行完后另外一个机器再执行，要么事件A先执行，要么事件B先执行，这样才能保证不会出现A:2 -> B:2 -> A:3 -> B:3这种积分越花越多的情况（想到这种bug一旦上线，老板要生气了，我可能要哭了）。

「怎么办？使用 zookeeper 分布式锁。」

一个机器接收到了请求之后，先获取 zookeeper 上的一把分布式锁（zk会创建一个 znode），执行操作；然后另外一个机器也**「尝试去创建」**那个 znode，结果发现自己创建不了，因为被别人创建了，那只能等待，等第一个机器执行完了方可拿到锁。

使用 ZooKeeper 的顺序节点特性，假如我们在/lock/目录下创建3个节点，ZK集群会按照发起创建的顺序来创建节点，节点分别为/lock/0000000001、/lock/0000000002、/lock/0000000003，最后一位数是依次递增的，节点名由zk来完成。

ZK中还有一种名为临时节点的节点，临时节点由某个客户端创建，当客户端与ZK集群断开连接，则该节点自动被删除。EPHEMERAL_SEQUENTIAL为临时顺序节点。

根据ZK中节点是否存在，可以作为分布式锁的锁状态，以此来实现一个分布式锁，下面是分布式锁的基本逻辑：

1. 客户端调用create()方法创建名为"/dlm-locks/lockname/lock-"的临时顺序节点。
2. 客户端调用getChildren("lockname")方法来获取所有已经创建的子节点。
3. 客户端获取到所有子节点path之后，如果发现自己在步骤1中创建的节点是所有节点中序号最小的，就是看自己创建的序列号是否排第一，如果是第一，那么就认为这个客户端获得了锁，在它前面没有别的客户端拿到锁。
4. 如果创建的节点不是所有节点中需要最小的，那么则监视比自己创建节点的序列号小的最大的节点，进入等待。直到下次监视的子节点变更的时候，再进行子节点的获取，判断是否获取锁。

释放锁的过程相对比较简单，就是删除自己创建的那个子节点即可，不过也仍需要考虑删除节点失败等异常情况。

5.面试官：ZK和Reids的区别，各自有什么优缺点？

「先说Reids：」

1. Rdis只保证最终一致性，副本间的数据复制是异步进行（Set是写，Get是读，Reids集群一般是读写分离架构，存在主从同步延迟情况），主从切换之后可能有部分数据没有复制过去可能会「丢失锁」情况，故强一致性要求的业务不推荐使用Reids，推荐使用zk。
2. Redis集群各方法的响应时间均为最低。随着并发量和业务数量的提升其响应时间会有明显上升（公有集群影响因素偏大），但是极限qps可以达到最大且基本无异常

「再说ZK：」

1. 使用ZooKeeper集群，锁原理是使用ZooKeeper的临时节点，临时节点的生命周期在Client与集群的Session结束时结束。因此如果某个Client节点存在网络问题，与ZooKeeper集群断开连接，Session超时同样会导致锁被错误的释放（导致被其他线程错误地持有），因此ZooKeeper也无法保证完全一致。
2. ZK具有较好的稳定性；响应时间抖动很小，没有出现异常。但是随着并发量和业务数量的提升其响应时间和qps会明显下降。

如何选择？（仅供参考，根据我个人经验）

关注指标	Redis	ZK
响应时间敏感	√	
并发量高	√	
需要读写锁		√
需要公平锁		√
需要非公平锁	√	

> **「提示」**
>
> 使用分布式锁，必须满足两个条件之一：
>
> 1. 业务本身不要求强一致性，可以接受偶尔出现锁被其他线程重复获取。
> 2. 业务本身要求强一致性，如果锁被错误地重复获取，必须有降级方案保证一致性。
>
> **「无论ZooKeeper与Redis，在极端情况下（例如整个ZK集群失效，例如Reids的Master失效而Slave没完全同步）都会存在正在被加锁的资源被重复加锁的问题。这种不可靠的概率极低，主要依赖于Zk集群与Redis集群。」**

6.Mysql如何做分布式锁？

「分布式锁还可以从数据库下手解决问题」

「方法一：」

利用 Mysql 的锁表，创建一张表，设置一个 UNIQUE KEY 这个 KEY 就是要锁的 KEY，所以同一个 KEY 在mysql表里只能插入一次了，这样对锁的竞争就交给了数据库，处理同一个 KEY 数据库保证了只有一个节点能插入成功，其他节点都会插入失败。

DB分布式锁的实现：通过主键id的唯一性进行加锁，说白了就是加锁的形式是向一张表中插入一条数据，该条数据的id就是一把分布式锁，例如当一次请求插入了一条id为1的数据，其他想要进行插入数据的并发请求必须等第一次请求执行完成后删除这条id为1的数据才能继续插入，实现了分布式锁的功能。

这样 lock 和 unlock 的思路就很简单了，伪代码：

```
def lock :
    exec sql: insert into locked-table (xxx) values (xxx)
    if result == true :
        return true
    else :
        return false

def unlock :
    exec sql: delete from lockedOrder where order_id='order_id'
```

「方法二：」

使用流水号+时间戳做幂等操作，可以看作是一个不会释放的锁。

7.面试官：你了解业界哪些大公司的分布式锁框架

「我：」 是时候展示我知识广度的时候了，这个B要装好哦

「1.Google:Chubby」

Chubby是一套分布式协调系统，内部使用Paxos协调Master与Replicas。

Chubby lock service被应用在GFS，BigTable等项目中，其首要设计目标是高可靠性，而不是高性能。

Chubby被作为粗粒度锁使用，例如被用于选主。持有锁的时间跨度一般为小时或天，而不是秒级。

Chubby对外提供类似于文件系统的API，在Chubby创建文件路径即加锁操作。Chubby使用Delay和SequenceNumber来优化锁机制。Delay保证客户端异常释放锁时，Chubby仍认为该客户端一直持有锁。Sequence number 指锁的持有者向Chubby服务端请求一个序号（包括几个属性），然后之后在需要使用锁的时候将该序号一并发给 Chubby 服务器，服务端检查序号的合法性，包括 number 是否有效等。

「2.京东SharkLock」

SharkLock是基于Redis实现的分布式锁。锁的排他性由SETNX原语实现，使用timeout与续租机制实现锁的强制释放。

「3.蚂蚁金服SOFAJRaft-RheaKV 分布式锁」

RheaKV 是基于 SOFAJRaft 和 RocksDB 实现的嵌入式、分布式、高可用、强一致的 KV 存储类库。

RheaKV对外提供lock接口，为了优化数据的读写，按不同的存储类型，提供不同的锁特性。RheaKV提供wathcdog调度器来控制锁的自动续租机制，避免锁在任务完成前提前释放，锁永不释放造成死锁。

「4.Netflix: Curator」

Curator是ZooKeeper的客户端封装，其分布式锁的实现完全由ZooKeeper完成。

在ZooKeeper创建EPHEMERAL_SEQUENTIAL节点视为加锁，节点的EPHEMERAL特性保证了锁持有者与ZooKeeper断开时强制释放锁；节点的SEQUENTIAL特性避免了加锁较多时的惊群效应。

总结

针对分布式锁的两种实现方法，使用哪种需要取决于业务场景，如果系统接口的读写操作完全是基于内存操作的，那显然使用Redis更合适，Mysql表锁or行锁明显不合适。同样是基于内存的 Redis锁 和 ZK锁具体选用哪一种，要根据是否有具体环境和架构师对哪种技术更为了解，原则就是选你最了解到，目的是能解决问题。

参考

Distributed locks with Redis

https://tech.meituan.com/2018/11/15/java-lock.html