前言

对多线程有所了解的朋友一般都会熟悉一个概念：锁。

在多线程并发场景下，要保证在同一时刻只有一个线程可以操作某个业务、数据或者变量，通常需要使用加锁机制。比如synchronized或Lock等。

而随着架构演进、业务发展，我们的应用往往都不是只部署在一台服务器上，而是使用分布式集群架构，同时存在多台相同的应用。

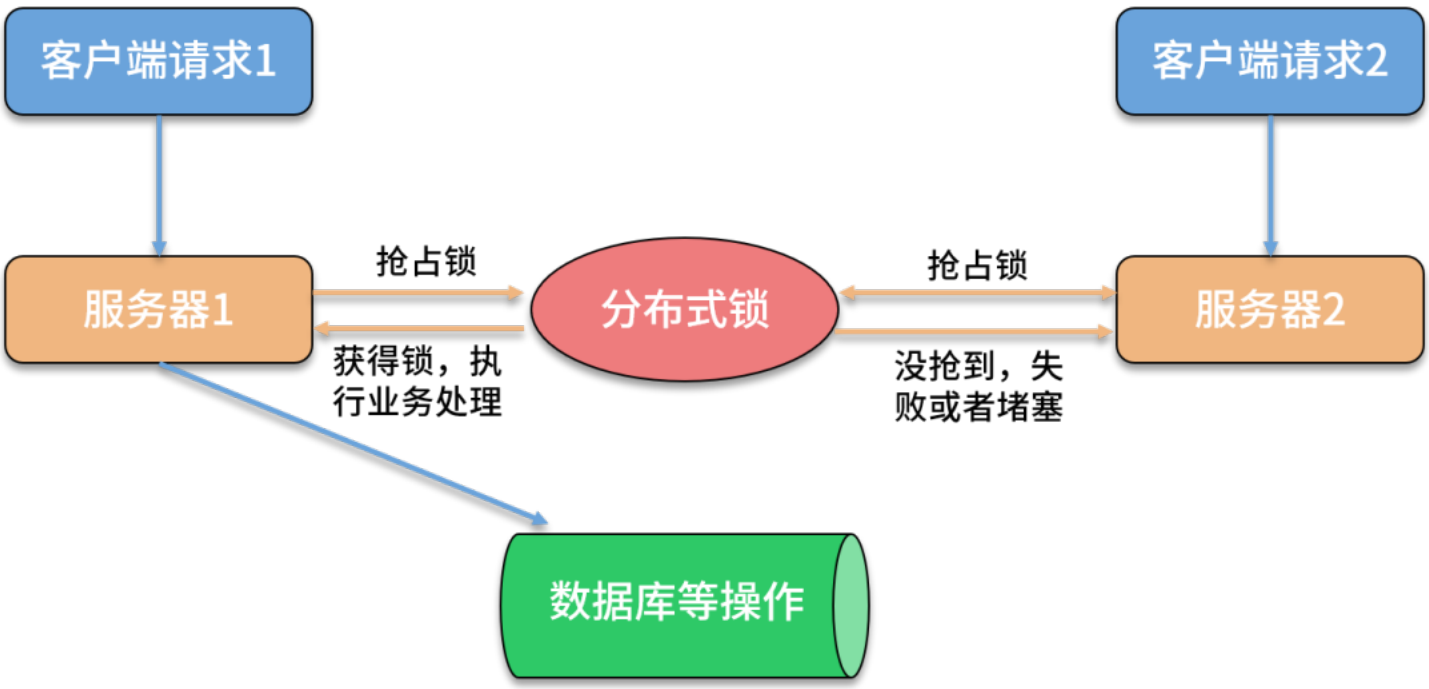
比如某电商网站在进行商品销售时，因为商品的数量是有限的，每个用户购买一件商品，需要将商品的库存减1；

但是我们想一下，在“双十一”这种火爆的活动时，可能会有大量的用户购买同一件商品，同时对该商品库存减1，如果不加锁，则极有可能会造成商品超卖情况。

要保证在这种分布式场景下，共享数据的安全性和一致性，则需要使用分布式锁。上面例子中的商品库存就是共享数据。

什么是分布式锁

顾名思义，分布式锁是指在分布式场景下，保证同一时刻对共享数据只能被一个应用的一个线程操作。用来保证共享数据的安全性和一致性。



分布式锁应该满足哪些要求

现在我们来分析一下，我们要实现一个分布式锁的话，需要满足哪些要求呢？

- 首先最基本的，我们要保证同一时刻只能有一个应用的一个线程可以执行加锁的方法，或者说获取到锁；（一个应用线程执行）
- 然后我们这个分布式锁可能会有很多的服务器来获取，所以我们一定要能够高性能的获取和释放；（高性能）
- 不能因为某一个分布式锁获取的服务不可用，导致所有服务都拿不到或释放锁，所以要满足高可用要求；（高可

用）

- 假设某个应用获取到锁之后，一直没有来释放锁，可能服务本身已经挂掉了，不能一直不释放，导致其他服务一直获取不到锁；（锁失效机制，防止死锁）
- 一个应用如果成功获取到锁之后，再次获取锁也可以成功；（可重入性）
- 在某个服务来获取锁时，假设该锁已经被另一个服务获取，我们要能直接返回失败，不能一直等待。（非阻塞特性）

以上是所有分布式锁要满足的一些基本要求。

实现方式有哪些

那么我们可以采取哪种方式来实现分布式锁呢？目前常见的方式主要有以下三种：

- 基于数据库实现
- 基于ZooKeeper实现
- 基于Redis实现

接下来我们看看这三种方式具体都是怎样实现分布式锁的。

基于数据库

使用数据库实现分布式锁，有两种方式。

第一种是基于数据库表实现。

比如我们有如下表来保存分布式锁记录：

```
CREATE TABLE `methodLock` (     `id` int(11) NOT NULL AUTO_INCREMENT COMMENT '主键',   `method_name` varchar(64) NOT NULL COMMENT '锁定的方法名',    `desc` varchar(1024) NOT NULL DEFAULT '备注信息',    `update_time` timestamp NOT NULL DEFAULT now() ON UPDATE now() COMMENT '保存时间',      PRIMARY KEY (`id`),    UNIQUE KEY `uidx_method_name` (`method_name `)) ENGINE=InnoDB COMMENT='分布式锁定的方法';
```

当我们的某个服务要执行某段需要分布式锁定的方法时，则执行插入语句，在该表中插入一条记录。

```
insert into methodLock(method_name,desc) values ('saleProduct','出售产品减库存');
```

因为我们在定义表时，method_name添加了唯一约束，如果在我们插入记录时，有多个服务都要执行这个操作，那么数据库可以保证只能有一个服务成功，我们认为只有插入成功的那个服务获取到了锁，可以继续执行该方法。

当方法执行完毕后，需要释放锁，则执行一条删除语句，将插入表中的记录删除。

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

另一种是基于数据库排他锁。

除了上面的通过插入删除的方式外，借助排它锁实现分布式锁。我们可以使用上面方法中的表，对于要使用分布式的方法提前插入一条记录。

```
insert into methodLock(method_name,desc) values ('saleProduct','出售产品减库存');
```

在某个应用线程需要对该方法加锁时，则使用如下语句：

```
select method_name from methodLock where method_name = 'saleProduct' for update
```

在查询语句后使用for update,数据库在查询时给该条记录添加上排他锁，其他线程则无法给该条记录添加锁。

我们可以当做查询到数据时，则获取到分布式锁，接下来执行方法中的逻辑。在方法执行完毕后，提交事务，锁会自动释放。

当然，还可以更简单一点，不用这么麻烦建一张表插入数据，而是对要进行分布式锁定的数据直接加锁。

比如我们要操作商品库存，在数据库中一般会有商品库存记录的表，比如叫：t_product_quantity，我们在对某商品减库存之前，先通过以下SQL查询出记录：

```
select product_no,quantity where product_no = xxx for update
```

同样该查询会对该产品的库存记录添加排它锁，之后其他线程都不可以对该条记录加锁。

接下来我们再对库存数据操作后，提交事务，锁会自动释放；如果操作过程中发生异常，事务回滚，也会自动释放锁。

使用以上基于数据库分布式锁的方式还是挺简单的。但是我们来回头看一下，这种方式是否能够满足我们上面列出来的分布式锁应该满足的要求呢？

- 一个应用一个线程执行 ✓
- 高性能&高可用
  - 因为是基于数据库实现的，所以高性能和高可用依赖于数据库，需要多机部署，主从同步、主备切换等。
- 失效机制
  - 需要手动删除，不具备失效机制。如果要支持失效机制，需要单独增加定时任务，按照记录的更新时间定时清除。
- 可重入性
  - 不具备，因为某线程在获取成功后，锁记录会一直存在，无法再次获取。
  - 可通过增加字段，记录占有锁的应用节点信息和线程信息，再次获取锁时判断是否是当前线程获取的锁达到可重入的特性。
- 非阻塞特性
  - 具备，在获取锁失败时，会直接返回失败。
  - 但是无法满足超时获取的场景，比如5秒内获取不到锁再失败等。

我们可以发现，这种方式虽然能满足最基本的分布式锁能力，但是在实际使用时，还是要针对一些问题做出优化，这些优化将会越来越复杂，并且存在一定的性能问题。所以一般**不建议基于数据库做分布式锁。**

基于ZooKeeper

基于ZooKeeper同样也能实现分布式锁，这里需要先铺垫一些ZK的基本知识。在ZK中，数据都是存放在数据节点中，数据节点称为Znode，ZK会将所有的数据都存放在内存中，所有的数据构成的数据模型是一个树状结构（ZNode Tree），不同层级的节点通过斜杠"/"分割，如/zoo/cat，和文件系统结构类似。

ZNode

在ZK中的数据节点分为以下四种：

持久节点

持久节点是ZK默认的节点类型，创建节点后，不管客户端与服务端是否断开连接，该节点会一直存在。

临时节点

可持久节点不同，临时节点在客户端与服务端断开连接后，临时节点会被删除。

顺序节点

顾名思义，顺序节点具有顺序，在创建节点时，ZK会根据创建时间给每个节点指定顺序编号。

临时顺序节点

临时顺序节点是临时节点和顺序节点的结合体，每个节点创建时会指定顺序编号，并且在客户端与ZK服务端断开时，节点会被删除。

ZK分布式锁实现原理

在ZK中并没有类似于Lock或Synchronized的API，它实现分布式锁依赖于**临时顺序节点**来完成。

获取锁

- 首先需要在ZK中先创建一个持久节点ParentLock表示一个分布式锁节点。
- 第一个客户端来获取锁时，就在这个ParentLock节点下创建一个顺序临时节点001-Node，然后查看ParentLock下所有临时顺序节点，判断当前创建节点是否在第一位，如果是，表示加锁成功；

- 之后第二个客户端来获取锁时，同样在ParentLock节点下创建一个顺序临时节点002-Node,然后判断自己是否在第一位，因为这是第一位是001-Node，所以这是会向排在自己前面的001-Node注册一个Watcher，用来监听001-Node节点，此时该客户端加锁失败，进入等待状态；

- 当有第三个客户端来时，同理因为新创建的003-Node不在第一位，于是向排在自己前面的002-Node注册一个Watcher，以此类推。

有没有发现，这里是形成了一个链式结构，和JUC中的AQS结构有点相似。

释放锁

释放锁的场景分两种，一种是业务处理完毕，正常释放锁；还有一种是客户端与服务端断开连接。

首先正常释放时，客户端会显式地将ZK中的数据节点删除；比如Client 1在业务处理完成时，将001-Node删除。

而客户端与服务器断开连接的情况，可能发生在客户端获取锁成功后，执行过程中发生异常，或应用崩溃，或网络异常等各种原因导致，这时ZK会自动将对应的Node节点删除。

由于Client 2一直在监听着001-Node节点，当001-Node节点删除后，Client 2会立刻收到通知，这时Client 2会再次查看节点列表，判断自己是否在最前面，如果是，则占有锁，表示加锁成功；

当Client 2释放锁之后，Client 3采用同样的方式处理。

以上就是使用ZooKeeper实现分布式锁的基本原理和过程。整体流程可以简化为下图所示。

要想在Java中使用ZK，官方有提供API包zkClient，使用时引入zookeeper-3.4.6.jar和 zkclient-0.1.jar即可；
也可使用第三方封装好的工具包，如Curator、Menagerie等。
通过以上我们可以看出，使用ZooKeeper实现分布式锁，基本可以全部满足我们对分布式锁的要求，需要注意的一点是，一定**要使用顺序临时节点，而不是临时节点**，使用临时节点会存在**羊群效应**问题。

基于Redis

使用Redis做分布式锁也是特别常见的一种选择。并且有多种实现方式。接下来我们逐个讲解。
常见的开源缓存组件都支持分布式锁，包括 Redis、Memcached 及 Tair。以常见的 Redis 为例，应用 Redis 实现分布式锁，最直接的想法是利用 setnx 和 expire 命令实现加锁。
在 Redis 中，setnx 是「set if not exists」如果不存在，则 SET 的意思，当一个线程执行 setnx 返回 1，说明 key 不存在，该线程获得锁；当一个线程执行 setnx 返回 0，说明 key 已经存在，那么获取锁失败，expire 就是给锁加一个过期时间。

第一种：SETNX+EXPIRE

这种方式可能是多数朋友第一反应就想到的，先通过SETNX获取到锁，然后通过EXPIRE命令添加超时时间。这种方式存在一个很大的问题，就是这两个命令的操作不是原子操作，需要和Redis交互两次，客户端可能会在第一个命令执行完之后就挂掉，导致没有设置上超时时间，那么这个锁就一直在那儿了。
为了解决这个问题，于是诞生了第二种方案。

第二种：SETNX+VALUE

这种方式的VALUE值中保存的是客户端计算出的过期时间，通过SETNX命令一次性放在Redis中；

```
public boolean getLock(String key,Long expireTime){    long expireTime = System.currentTimeMills()+expireTime;    String value = String.valueOf(expireTime);    // 加锁成功    if(jedis.setnx(key,value)==1){        return true;    }    // 获取锁的value    String currentValueStr = jedis.get(key);    // 如果过期时间小于系统时间，则表示已过期    if (currentValueStr != null && Long.parseLong(currentValueStr) < System.currentTimeMillis()) {        // 锁已过期，获取上一个锁的过期时间，并设置现在锁的过期时间        String oldValueStr = jedis.getSet(key_resource_id, expiresStr);        if (oldValueStr != null && oldValueStr.equals(currentValueStr)) {            // 考虑多线程并发的情况，只有一个线程的设置值和当前值相同，它才可以加锁            return true;        }    }    //其他情况，均返回加锁失败    return false;}
```

这种方式通过value将超时时间赋值，解决了第一种方案的两次操作不原子性的问题。但是这种方式也有问题：

- 在锁过期时，如果多个线程同时来加锁，可能会导致多个线程都加锁成功；
- 当多个线程都加锁成功后，因为锁中没有加锁线程的标识，会导致多个线程都可以解锁；
- 超时时间是在客户端计算的，不同的客户端的时钟可能会存在差异，导致在加锁客户端没有超时的锁，在另一个客户端已经超时。

第三种：使用Lua脚本

同样是为了解决第一种方案中的原子性问题，我们可以采用Lua脚本，来保证SETNX+EXPIRE操作的原子性。

```
if redis.call('setnx',KEYS[1],ARGV[1]) == 1 then
redis.call('expire',KEYS[1],ARGV[2])else    return 0end;
```

在Java代码中，使用jedis.eval()执行加锁。

```
public boolean getLock(String key,String value)
{    String lua_scripts = "if redis.call('setnx',KEYS[1],ARGV[1]) == 1 "        + "then redis.call('expire',KEYS[1],ARGV[2]) return 1 else return 0 end";
 Object result = jedis.eval(lua_scripts,Collections.singletonList(key),Collections.singletonList(value));
  return result.equals(1L);}
```

这种方式可以完全避免在加锁后中断设置不上超时时间的问题。也不会存在有时钟不一致的问题，和高并发情况下多个线程都加上锁的问题。但是这种方式就一定没有问题了吗？答案是否定的，看下图。

当服务A加锁成功后，正在执行业务的过程中，锁过期啦，这时服务A是没有感知的；
接着服务B这时来获取锁，成功获取到了；
紧接着，服务A处理完业务了，来释放锁，成功释放掉了，而服务B这时还以为它的锁还在，在执行代码。
全乱套了有没有？以为自己加锁了，其实你没加；
以为自己解锁成功了，其实解的是别人的锁；

这种方案的问题主要是因为两点：**锁过期释放，业务没处理完；锁没有唯一身份标识。**

第四种：SET NX PX EX + 唯一标识

对于误删锁的问题，我们可以在加锁时，由客户端生成一个唯一ID作为value设置在锁中，在删除锁时先进行身份判断，再删除；加锁逻辑如下：

```
public boolean getLock(String key,String uniId,Long expireTime){    //加锁    return jedis.set(key, uniId, "NX", "EX", expireTime) == 1;}// 解锁public boolean releaseLock(String key,String uniId){    // 因为get和del操作并不是原子的，所以使用lua脚本  String lua_script = "if redis.call('get',KEYS[1]) == ARGV[1] then  return redis.call('del',KEYS[1])         +"else return 0  end;";    Object result = jedis.eval(lua_scripts,Collections.singletonList(key),Collections.singletonList(uniId));    return result.equals(1L);}
```

这种方式解决了锁被误删的问题，但是**同样存在锁超时失效，但是业务还未处理完的问题。**

第五种：Redission框架

那么对于锁过期失效，业务未处理完毕的问题，该如何处理呢？

我们可以在加锁成功后，启动一个守护线程，在守护线程中隔一段时间就对锁的超时时间再续长一点，直到业务处理完成后，释放锁，防止锁在业务处理完毕之前提前释放。

而Redission框架就是使用的这种机制，来解决的这个问题。

当一个线程去获取锁，在加锁成功的情况下，那么它已经同Lua脚本将数据保存在了redis中；
然后在加锁成功的同时，启动watch Dog看门狗，每隔10秒检查是否还持有锁，如果是则将锁超时时间延长。
如果一开始就获取锁失败，则会一直循环获取。
这种方案的Redis锁总该没有问题了吧？格局小了呀我滴朋友，还有问题呢。

以上的这些方案，都只是在Redis单机模式下讨论的方案，如果Redis是采用集群模式，还会存在一些问题，不过问题不是很离谱，我们来简单讲解一下。

在集群模式下，一般Master节点会将数据同步到Salve节点，如果我们现在Master节点上加锁成功，在同步到Salve节点之前，这个Master节点挂了，然后另一台Salve节点升级为Master节点，这时这个节点上并没有我们的加锁数据；

此时另一个客户端线程来获取相同的锁，它就会获取成功，这时在我们的应用中将会有两个线程同时获取到这个锁，这个锁也就不安全了。

为了解决这个问题，Redis的作者亲自出马了，提出了一种高级的分布式锁算法，很牛批，叫：**RedLock。**

第六种：RedLock+Redi

首先这个RedLock的意思并不是“红色的锁”，和红色没啥关系，而是Redis Distributed Lock,Redis分布式锁，看见没，这才是正主，官方出品。

RedLock的核心原理是这样的：

- 在Redis集群中选出多个Master节点，保证这些Master节点不会同时宕机；
- 并且各个Master节点之间相互独立，数据不同步；
- 使用与Redis单实例相同的方法来加锁和解锁。

那么RedLock到底是如何来保证在有节点宕机的情况下，还能安全的呢？

1. 假设集群中有N台Master节点，首先，获取当前时间戳；
2. 客户端按照顺序使用相同的key,value依次获取锁，并且获取时间要比锁超时时间足够小；比如超时时间5s,那么获取锁时间最多1s，超过1s则放弃，继续获取下一个；
3. 客户端通过获取所有能获取的锁之后减去第一步的时间戳，这个时间差要小于锁超时时间，并且要至少有N/2 + 1台节点获取成功，才表示锁获取成功，否则算获取失败；
4. 如果成功获取锁，则锁的有效时间是原本超时时间减去第三不得时间差；
5. 如果获取锁失败，则要解锁所有的节点，不管该节点加锁时是否成功，防止有漏网之鱼。

Redssion库对RedLock方案已经做了实现，如果你的Redis是集群部署，可以看看使用方法。

> 参考文档：https://redis.io/topics/distlock

通过以上6种基于Redis的方式，我们该如何选择呢？可以按以下原则：

- 如果Redis是单机部署，使用方案五，Redission框架，加锁时可按场景开启watch dog机制；解锁时使用Lua脚本原子删除；
- 如果是集群部署，建议采用RedLock方案实现。