分布式锁

今天分享的主题是分布式锁。

动机：一致性问题

在任何分布式系统中，只要出现并发，就一定有一致性问题。这里举一个简单的例子，买家在电商平台上购买了一件商品，卖家还未发货；此时买家发出了取消订单的请求，同时卖家发出了发货的请求，那么平台应该怎么处理？暂且不论正确结果应该是什么，至少平台不能简单地直接并发处理两个请求，否则两个请求都收到了成功的回复，而订单不可能同时在发货和取消状态。

这种例子是非常常见的。分布式数据库的数据复制、分布式资源的竞争访问、甚至是单机并发程序的内存访问。处理这些问题并不一定要用分布式锁，但分布式锁是一个直观的方案，而且在某些场景下难以被替代。

模型和假设

按照惯例，我们仍然假设应用的数据部分是一个 key-value 映射表（也被称为寄存器），支持以下操作：

```
write(k, v)
read(k) → v
```

涉及到加锁，扩展两条指令：

```
lock(k)
unlock(k)
```

加锁

在订单的例子中，一个订单同时只能处于一种状态，且可响应的请求和状态相关。要避免上述不一致的结果发生，最简单的方式就是为订单关联一个锁。只有持有锁的客户端才能继续访问订单，其它并发客户端则会阻塞在获取锁阶段或者获取锁失败。

这种抽象的「锁」机制有三种常见的实现方式：数据库锁、分区应用和分布式锁。

数据库锁

数据库锁最常用，但不是本次分享的重点，因此作简要介绍。

假设所有要处理的数据都在同一个数据库中，直接利用数据库的并发控制机制可以简单地解决并发问题。

悲观锁

数据库锁是数据库系统的机制，为了防止某些记录被并发修改。例如以下 SQL 语句：

```
SELECT ... LOCK IN SHARE MODE

SELECT ... FOR UPDATE
```

就会为相应的行加锁（有时也可能锁住多行或者整个表）。第一行是共享锁，第二行是排他锁。共享锁允许多个共享锁一同持有，而排他锁不能和其它锁共存。当某些行需要被更新时，就会获得排他锁，而读数据只需要获取共享锁。

悲观锁有潜在的死锁问题和性能问题。性能问题来自排他锁阻止了并发访问，而死锁问题可以通过下面的转账例子来理解：

```
def trans(id1, id2, amount):
  lock(id1)
  lock(id2)
  balance1 = read(id1)
  balance2 = read(id2)
  if balance1 ≥ amount:
    write(id1, balance1 - amount)
    write(id2, balance2 + amount)
    unlock(id1)
    unlock(id2)
    return Ok
  else:
    unlock(id1)
    unlock(id2)
    return Err
```

考虑 A 在给 B 转账的同时，B 在给 A 转账。此时两个并发的处理过程都执行到了第二行语句，也就是 A 和 B 两个账户都加锁了，而第三行语句都无法执行，造成死锁。预防和处理死锁的办法也有不少，例如给所有 id 排序，然后按照顺序加锁。但死锁的潜在问题仍然是存在的，而且不可能一劳永逸地解决（此处不作详细证明）。

乐观锁

乐观锁也叫乐观并发控制，思路也很简单，以以下 SQL 语句为例：

```
1.
SELECT id, data, version
FROM table
WHERE id = $ID

2.
Business Code

3.
UPDATE table
SET data = $data
    version = version + 1
WHERE id = $ID AND version = $VER

4.
if UPDATE failed, go to step 1
```

在修改任何行之前，获取其版本号，并且在修改提交时更新版本号。如果提交时该行的版本号发生了变化，说明并发修改发生了，此时就直接让修改过程失败。

乐观锁的实现比悲观锁复杂，需要客户端参与（侵入性）。

分区应用

略。

分布式锁 DLM

考虑更宽泛的场景，例如要实现分布式事务的原子提交，事务涉及到多对象的读写，而不同对象存储于不同的数据库服务中。此时无法使用数据库的并发控制。大体上来说，需要一种分布式锁服务来协调并发任务。这个分布式锁具备以下功能：

- 提供加锁、解锁协议
- 高可用，意味着没有单点故障并且持久化
- 死锁预防，通常使用过期机制或者心跳检测

单纯实现 `lock`，`unlock` 操作是很容易的。可以设想实现这样一个服务，其主存的内容就是一个 key-value 映射表，value 部分是 `Mutex<Option<Client>>`，用于存储持有该 key 的客户端 id。当 value 为空就表示这个 key 没有被加锁。Mutex 是内存中的锁，用于保证 value 不会被两个线程同时修改。

困难的部分在于高可用。注意，在实现单机锁服务时使用了内存锁，而多节点想要同时锁住一个对象该怎么做呢？没错，分布式锁服务。因此分布式锁在实现高可用时一定需要一套严密的机制。

死锁的检测或预防比单机程序的锁更加复杂，因为分布式应用的各个组件都可能是不可靠的。一个服务可能获取锁后就永久掉线了，如果没有对应的机制来释放锁，这个锁就会被永久持有，这也是死锁的一种情况。

Redis

观察 Redis 这样的一条加锁操作：

```
SET resource_name my_random_value NX PX 30000
```

语句的意思是，如果 `resouce_name` 这个 key 不存在，就将其值设置为 `my_random_value`。过期时间是30秒，30秒后无论有没有主动释放，该 key 都会自动被删除。

解锁的操作时这样的：

```
if redis.call("get", KEYS[1]) == ARGV[1] then
  redis.call("del", KEYS[1])
end
```

Redis 保证了 lua 脚本会原子执行。

因此 Redis 可以直接充当 DLM。

Redis redlock

要实现高可用，Redis 的做法是在客户端实现一套算法，这个算法叫作 redlock。算法描述如下：

1. 客户端获取当前时间，精确到毫秒
2. 客户端对 Redis 集群的所有示例挨个执行上述加锁操作，使用相同的 key 和随机数 value
3. 如果这个加锁的总时间小于锁的有效时间（上述过期时间），并且超过一半的示例都获取锁成功，则认为加锁操作成功
4. 对每个实例的加锁有效时间为 `validity time = initial validity time - elapsed time`
5. 如果一个实例的加锁过程失败，则整个加锁过程失败，释放掉所有已经加的锁

用一个例子来解释上述算法。假设现在有5个 Redis 实例，客户端想要获取 key 为 k-1 的锁，超时时间为30000毫秒。客户端的操作流程如下：

1. 获取当前时间，例如为10000
2. 生成唯一的随机数，例如12345
3. 开始尝试对 Redis-0 加锁，执行 `SET k-1 12345 NX PX 30000`，执行成功
4. 再次获取时间，由于上述操作经历了2000ms，所以现在的时间是12000

5. 尝试对 Redis-1 加锁，执行 `SET k-1 12345 NX PX 28000`，执行成功。新的有效时间计算就是为了多个实例能同时释放锁
6. 重复上述过程，对 Redis-2 加锁，执行成功
7. 整个加锁过程宣告成功，接下来异步地对 Redis-3 和 Redis-4 加锁

这个算法一眼能看出很多问题。首先，所有逻辑都实现在客户端，依赖客户端的正确行为；其次，同时依赖客户端的当前时间和 Redis 服务器的时间，局部时间在分布式系统中是非常不可靠的。下面具体说明该算法的问题。

Redis redlock review

对于锁服务的互斥性的要求是非常苛刻的。一旦一个锁同时被两个客户端持有，可能会发生非常糟糕的事情，例如数据库的两个副本都认为自己是主副本（脑裂）。而对于过期时间或者响应延迟则没有过于严格的要求。

考虑这样的场景：

1. Redis 在 A、B、C、D、E 五个节点上运行，现有两个客户端 Client1 和 Client2 要获取同一个锁
2. Client1 按 A、B、C 的顺序获取锁，Client2 按 E、D、C 的顺序获取锁
3. Client1 已经成功在 A、B 获取锁，Client2 成功在 E、D 获取锁
4. Client1 在 C 获取锁，超时时间是30s
5. 但节点 C 突然发生了时间跳跃（这是可能发生的，无论是网络同步还是人为修改），导致节点 C 释放了锁
6. Client2 在 C 获取锁

此时 Client1 和 Client2 都认为自己获得了锁，破坏了互斥性。

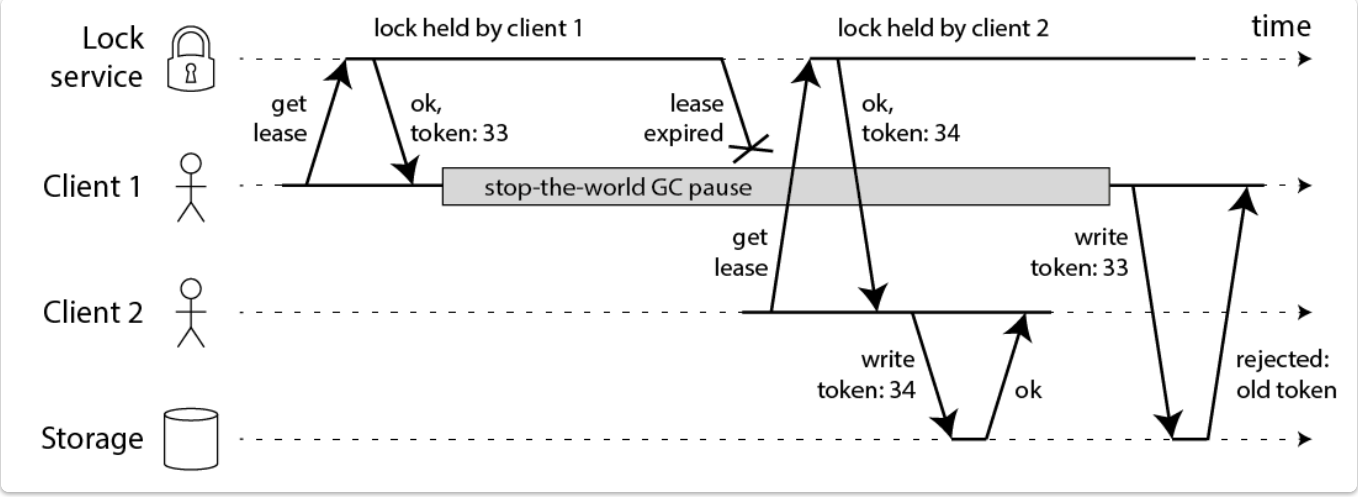
考虑第二个场景，客户端在成功完成多数节点的加锁后发生了进程暂停。进程暂停经常发生在带有 GC 的语言中，这些语言的运行时可能会暂停整个世界来进行 GC。即便没有 GC，机器性能和进程调度也可能导致进程暂停。当客户端暂停结束，以为自己仍然持有锁，但实际上进程暂停了一分钟，锁已经过期了，这期间由于进程暂停客户端也丢失了关于锁过期的通知。

考虑第三个场景，客户端和某个节点之间有非常严重的网络延迟，这个延迟可能有一分钟，也可能使得客户端以为自己持有锁，而锁实际上被释放了。

后面两个问题看似可以通过延长租期时间来解决，但问题是我们不知道这个时间应当被延长为多少。

Fencing

Martin Kleppmann 认为如果一定要用这种方式实现分布式锁，至少还需要添加如下 fencing 机制：



成功获取锁的同时还会获取一个自增的 token 值，而只有最新的 token 值被允许访问互斥资源。

这个方案的问题在于分布式锁的实现逻辑被同时生成在了 DLM、客户端和互斥资源处。这个方案最终实现的其实更像乐观锁，回顾我们在乐观锁部分谈到的版本号机制。那么问题就来了，既然可以使用乐观锁，DLM 的意义在哪?

ZooKeeper

使用 ZooKeeper 实现共享锁和排他锁也是可能的，下面是操作方法：

1. 创建 `_locknode/lock-`，附带 Sequence 和 Ephemeral 标签
2. 获取 children，检查自己是不是最小的数字
3. 如果自己是最小的，则获得锁；否则利用 watch 机制进行等待

Sequence 标签意味着多次对于该 node 的创建会产生顺序自增的名字。Ephemeral 标签意味着当 Client 失去心跳（断联），这个 node 就会被删除。

ZooKeeper 自身能保证强一致性、高可用。用 ZooKeeper 实现分布式锁是相当实用的选择。

这种可线性化的 CAS 操作的实现依赖共识算法。ZooKeeper 使用了 zab 算法，而 etcd 使用了 raft。ZooKeeper 可以看做是 Google Chubby 的开源社区版，Chubby 本身使用 Pasox 算法。

总结

1. 数据库的锁足以自动应付数据的并发访问。
2. 乐观锁没有死锁风险，大部分情况下性能更好。如果应用和数据之间有乐观锁协议，那么锁服务就是不需要的。
3. 分布式锁服务很难开发，尤其要考虑到高可用。
4. 高可用性意味着容错和数据复制，容错意味着选举，这些都依赖共识算法。
5. 分布式锁的死锁预防常用租期或心跳实现。

References

https://martin.kleppmann.com/2016/02/08/how-to-do-distributed-locking.html

本文的 Redis redlock review 和 Fencing 部分的内容来自于这篇文章。作者是 Martin Kleppmann。

《数据密集型应用系统设计》第8、9章

作者也是 Martin Kleppmann。该书对数据系统的各个方面作了非常深入的介绍。

https://redis.io/topics/distlock/

Redis 的分布式锁算法和其 Redlock 算法。

https://zookeeper.apache.org/doc/r3.1.2/recipes.html

关于 ZooKeeper 的一切功能的使用方式。