为什么需要锁

原因其实很简单：因为我们想让同一时刻只有一个线程在执行某段代码。

因为如果同时出现多个线程去执行，可能会带来我们不想要的结果，可能是数据错误，也可能是服务宕机等等。

以淘宝双11为例，在0点这一刻，如果有几十万甚至上百万的人同时去查看某个商品的详情，这时候会触发商品的查询，如果我们不做控制，全部走到数据库去，那是有可能直接将数据库打垮的。

这个时候一个比较常用的做法就是进行加锁，只让1个线程去查询，其他线程待等待这个线程的查询结果后，直接拿结果。在这个例子中，锁用于控制访问数据库的流量，最终起到了保护系统的作用。

再举个例子，某平台做活动"秒杀茅台"，假如活动只秒杀1瓶，但是同时有10万人在同一时刻去抢，如果底层不做控制，有10000个人抢到了，额外的9999瓶平台就要自己想办法解决了。此时，我们可以在底层通过加锁或者隐式加锁的方式来解决这个问题。

此外，锁也经常用来解决并发下的数据安全方面的问题，这里就不一一举例了。

为什么需要分布式锁

分布式锁是锁的一种，通常用来跟 JVM 锁做区别。

JVM 锁就是我们常说的 synchronized、Lock。

JVM 锁只能作用于单个 JVM，可以简单理解为就是单台服务器（容器），而对于多台服务器之间，JVM 锁则没法解决，这时候就需要引入分布式锁。

实现分布式锁的方式

实现分布式锁的方式其实很多，只要能保证对于抢夺"锁"的系统来说，这个东西是唯一的，那么就能用于实现分布式锁。

举个简单的例子，有一个 MySQL 数据库 Order，Order 库里有个 Lock 表只有一条记录，该记录有个状态字段 lock_status，默认为0，表示空闲状态，可以修改为1，表示成功获取锁。

我们的订单系统部署在100台服务器上，这100台服务器可以在“同一时刻”对上述的这1条记录执行修改，修改内容都是从0修改为1，但是 MysQL 会保证最终只会有1个线程修改成功。因此，这条记录其实就可以用于做分布式锁。

常见实现分布式锁的方式有：数据库、Redis、Zookeeper。

这其中又以 Redis 最为常见。

Redis 实现分布式锁

加锁

加锁通常使用 set 命令来实现，伪代码如下：

```
set key value PX milliseconds NX
```

几个参数的意义如下：

key、value：键值对

PX milliseconds：设置键的过期时间为 milliseconds 毫秒。

NX：只在键不存在时，才对键进行设置操作。SET key value NX 效果等同于 SETNX key value。

PX、expireTime 参数则是用于解决没有解锁导致的死锁问题。因为如果没有过期时间，万一程序员写的代码有 bug 导致没有解锁操作，则就出现了死锁，因此该参数起到了一个“兜底”的作用。

NX 参数用于保证在多个线程并发 set 下，只会有1个线程成功，起到了锁的“唯一”性。

解锁

解锁需要两步操作：

1）查询当前“锁”是否还是我们持有，因为存在过期时间，所以可能等你想解锁的时候，“锁”已经到期，然后被其他线程获取了，所以我们在解锁前需要先判断自己是否还持有“锁”

2）如果“锁”还是我们持有，则执行解锁操作，也就是删除该键值对，并返回成功；否则，直接返回失败。

由于当前 Redis 还没有原子命令直接支持这两步操作，所以当前通常是使用 Lua 脚本来执行解锁操作，Redis 会保证脚本里的内容执行是一个原子操作。

脚本代码如下，逻辑比较简单：

```
if redis.call("get",KEYS[1]) == ARGV[1]
then
    return redis.call("del",KEYS[1])
else
    return 0
end
```

两个参数的意义如下：

KEYS[1]：我们要解锁的 key

ARGV[1]：我们加锁时的 value，用于判断当“锁”是否还是我们持有，如果被其他线程持有了，value 就会发生变化。

上述方法是 Redis 当前实现分布式锁的主流方法，可能会有一些小优区别，但是核心都是这个思路。看着好像没啥毛病，但是真的是这个样子吗？让我们继续往下看。

Redis 分布式锁过期了，还没处理完怎么办

为了防止死锁，我们会给分布式锁加一个过期时间，但是万一这个时间到了，我们业务逻辑还没处理完，怎么办？

首先，我们在设置过期时间时要结合业务场景去考虑，尽量设置一个比较合理的值，就是理论上正常处理的话，在这个过期时间内是一定能处理完毕的。

之后，我们再来考虑对这个问题进行兜底设计。

关于这个问题，目前常见的解决方法有两种：

1、守护线程“续命”：额外起一个线程，定期检查线程是否还持有锁，如果有则延长过期时间。Redisson里面就实现了这个方案，使用“看门狗”定期检查（每1/3的锁时间检查1次），如果线程还持有锁，则刷新过期时间。

2、超时回滚：当我们解锁时发现锁已经被其他线程获取了，说明此时我们执行的操作已经是“不安全”的了，此时需要进行回滚，并返回失败。

同时，需要进行告警，人为介入验证数据的正确性，然后找出超时原因，是否需要对超时时间进行优化等等。

守护线程续命的方案有什么问题吗

Redisson 使用看门狗（守护线程）“续命”的方案在大多数场景下是挺不错的，也被广泛应用于生产环境，但是在极端情况下还是会存在问题。

问题例子如下：

1、线程1首先获取锁成功，将键值对写入 redis 的 master 节点

2、在 redis 将该键值对同步到 slave 节点之前，master 发生了故障

3、redis 触发故障转移，其中一个 slave 升级为新的 master

4、此时新的 master 并不包含线程1写入的键值对，因此线程2尝试获取锁也可以成功拿到锁

5、此时相当于有两个线程获取到了锁，可能会导致各种预期之外的情况发生，例如最常见的脏数据

解决方法：上述问题的根本原因主要是由于 redis 异步复制带来的数据不一致问题导致的，因此解决的方向就是保证数据的一致。

当前比较主流的解法和思路有两种：

1）Redis 作者提出的 RedLock；2）Zookeeper 实现的分布式锁。

接下来介绍下这两种方案。

RedLock

首先，该方案也是基于文章开头的那个方案（set加锁、lua脚本解锁）进行改良的，所以 antirez 只描述了差异的地方，大致方案如下。

假设我们有 N 个 Redis 主节点，例如 N = 5，这些节点是完全独立的，我们不使用复制或任何其他隐式协调系统，为了取到锁，客户端应该执行以下操作:

1、获取当前时间，以毫秒为单位。

2、依次尝试从5个实例，使用相同的 key 和随机值（例如UUID）获取锁。当向Redis 请求获取锁时，客户端应该设置一个超时时间，这个超时时间应该小于锁的失效时间。例如你的锁自动失效时间为10秒，则超时时间应该在 5-50 毫秒之间。这样可以防止客户端在试图与一个宕机的 Redis 节点对话时长时间处于阻塞状态。如果一个实例不可用，客户端应该尽快尝试去另外一个Redis实例请求获取锁。

3、客户端通过当前时间减去步骤1记录的时间来计算获取锁使用的时间。当且仅当从大多数（N/2+1，这里是3个节点）的Redis节点都取到锁，并且获取锁使用的时间小于锁失效时间时，锁才算获取成功。

4、如果取到了锁，其有效时间等于有效时间减去获取锁所使用的时间（步骤3计算的结果）。

5、如果由于某些原因未能获得锁（无法在至少N/2+1个Redis实例获取锁、或获取锁的时间超过了有效时间），客户端应该在所有的Redis实例上进行解锁（即便某些Redis实例根本就没有加锁成功，防止某些节点获取到锁但是客户端没有得到响应而导致接下来的一段时间不能被重新获取锁）。

可以看出，该方案为了解决数据不一致的问题，直接舍弃了异步复制，只使用 master 节点，同时由于舍弃了 slave，为了保证可用性，引入了 N 个节点，官方建议是 5。

该方案看着挺美好的，但是实际上我所了解到的在实际生产上应用的不多，主要有两个原因：1）该方案的成本似乎有点高，需要使用5个实例；2）该方案一样存在问题。

该方案主要存以下问题：

1）严重依赖系统时钟。如果线程1从3个实例获取到了锁，但是这3个实例中的某个实例的系统时间走的稍微快一点，则它持有的锁会提前过期被释放，当他释放后，此时又有3个实例是空闲的，则线程2也可以获取到锁，则可能出现两个线程同时持有锁了。

2）如果线程1从3个实例获取到了锁，但是万一其中有1台重启了，则此时又有3个实例是空闲的，则线程2也可以获取到锁，此时又出现两个线程同时持有锁了。

针对以上问题其实后续也有人给出一些相应的解法，但是整体上来看还是不够完美，所以目前实际应用得不是那么多。

Zookeeper 实现分布式锁

Zookeeper 的分布式锁实现方案如下：

1、创建一个锁目录 /locks，该节点为持久节点

2、想要获取锁的线程都在锁目录下创建一个临时顺序节点

3、获取锁目录下所有子节点，对子节点按节点自增序号从小到大排序

4、判断本节点是不是第一个子节点，如果是，则成功获取锁，开始执行业务逻辑操作；如果不是，则监听自己的上一个节点的删除事件

5、持有锁的线程释放锁，只需删除当前节点即可。

6、当自己监听的节点被删除时，监听事件触发，则回到第3步重新进行判断，直到获取到锁。

由于 Zookeeper 保证了数据的强一致性，因此不会存在之前 Redis 方案中的问题，整体上来看还是比较不错的。

Zookeeper 方案的主要问题在于性能不如 Redis 那么好，当申请锁和释放锁的频率较高时，会对集群造成压力，此时集群的稳定性可用性能可能又会遭受挑战。

分布式锁的选型

当前主流的方案有两种：

1）Redis 的 set 加锁+lua 脚本解锁方案，至于是不是用守护线程续命可以结合自己的场景去决定，个人建议还是可以使用的。

2）Zookeeper 方案

通常情况下，对于数据的安全性要求没那么高的，可以采用 Redis 的方案，对数据安全性要求比较高的可以采用 Zookeeper 的方案。