Redis 分布式锁解决方案

tojson 老师

一、分布式锁概念

在 Java 中，关于锁我想大家都很熟悉。在并发编程中，我们通过锁来解决由于线程竞争而造成的数据不一致问题。通常，不同的场景我们会使用不同的锁：

- JVM 锁（单机）

Java 中的锁，只能保证在同一个 JVM 进程内中多线程并发访问资源的安全性，一般使用 JDK 自带的 synchronized 或 lock 锁。

- 分布式锁（集群）

用来解决在集群环境下不同进程多线程并发访问资源的安全性。一般采用 mysql、zk、redis 实现。

二、分布式锁经典应用场景-商品秒杀超卖

- 商品秒杀超卖

常见解决方案：

1、悲观锁

当查询某条记录时，就让数据库为该记录加锁，锁住记录后别人无法操作。

一般提前采用 select for update，提前加上锁。不过并发环境下容易出现死锁。

当减库存和高并发读写碰到一起的时候，由于操作的库存数目在同一行，就会出现争抢 InnoDB 行锁的问题，导致出现互相等待甚至死锁，从而大大降低 mysql 的处理性能。

缺点：可能出现死锁，并发性能差。

2、乐观锁

由于悲观锁容易出现死锁，所以引入了乐观锁方案，乐观锁并不是真实存在的锁，而是在更新的时候判断此时的库存是否是之前查询出的库存，如果相同，表示没人修改，可以更新库存，否则表示别人抢过资源，不再执行库存更新。

采用乐观锁需修改数据库的事务隔离级别：

我们在使用乐观锁的时候，如果一个事务修改了库存并提交了事务，那其他的事务应该可以读取到修改后的数据值，所以不能使用数据库默认的事务隔离级别可重复读的隔离级别 RR，应该修改为读已提交的隔离级别（Read committed）。

缺点：并发性能差。

3、redis 分段预扣库存和异步消息写入 db

分两个阶段完成：

申请阶段：

将库存扣减从 mysql 前移到 Redis 中，所有的预减库存的操作放到 redis 中，由于 Redis 中不存在锁，因此不会出现互相等待，并且由于 Redis 的写性能和读性能都远高于 mysql，这就解决了高并发下的性能问题。为了提高并发能力，一般对 redis 锁进行分段处理。

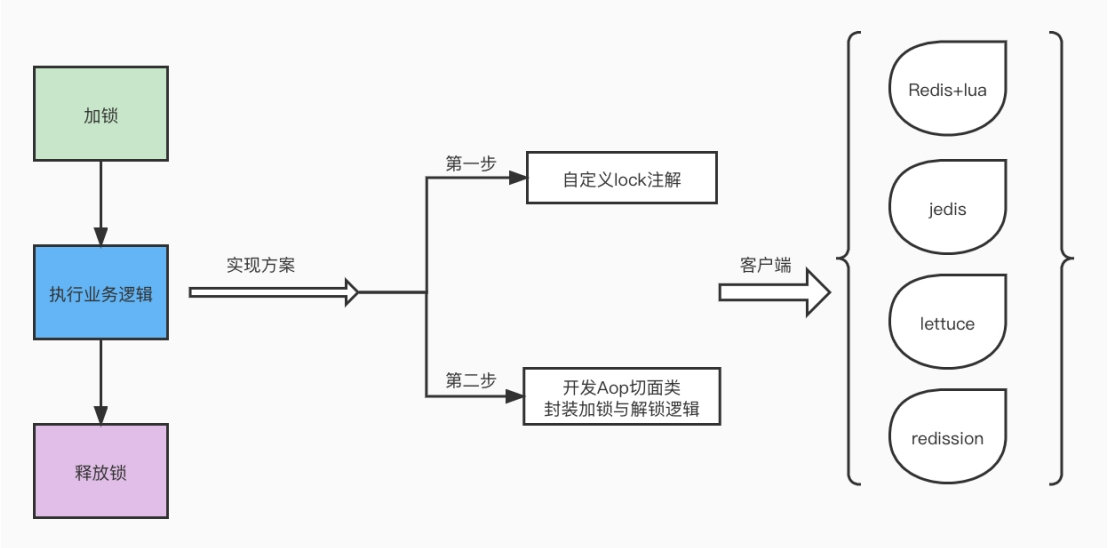
确认阶段：

通过 mq 等异步手段，将变化的数据异步写入到 DB 中。引入 mq，数据通过 mq 排序，按照次序更新到 DB 中，完全串行处理。当达到库存阀值的时候就不在消费队列，并关闭购买功能，这样就解决了商品超卖问题。

缺点：由于异步写入 DB，可能存在数据不一致的风险，所以需要设计相应的告警机制和补偿措施。

三、分布式锁实现

整体流程图如下：



1）加锁

2）执行业务逻辑

3）释放锁

springboot 实现方式：

自定义分布式注解@lock + AOP（开发一个切面拦截类封装加锁与解锁逻辑）

引入 spring-data-redis 驱动包，集成 redis 某个客户端

redis 客户端：

- Redis + Lua
- jedis 客户端
- lettuce 客户端
- redission 客户端

四、常见大厂面试题

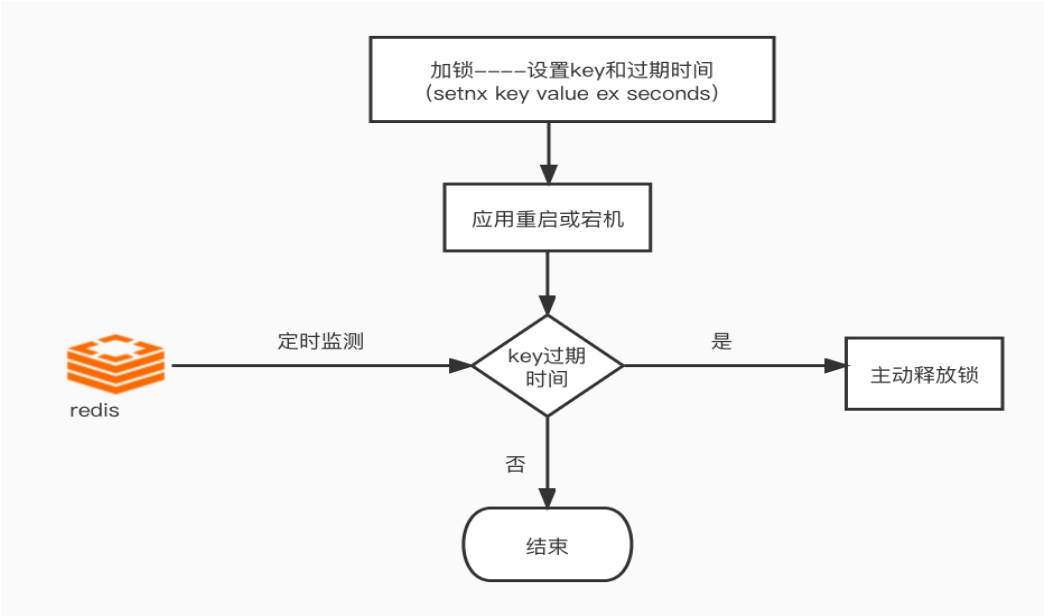
1）redis 除了做缓存，还可以用来做什么？

答：分布式锁、防重提交、分布式限流、简易版本的消息队列、延迟任务、session 共享（集成 spring-session-data-redis）

2）redis 锁如何保证任何时刻有且只有一个线程持有这个锁？

答：使用命令：setnx key value　key 不存在时设置成功返回值 ok，key 存在设置失败，也可以采用 If(!redisUtil.get(key)){set key value}，不过这段代码需要采用 lua 脚本实现来保证原子性。

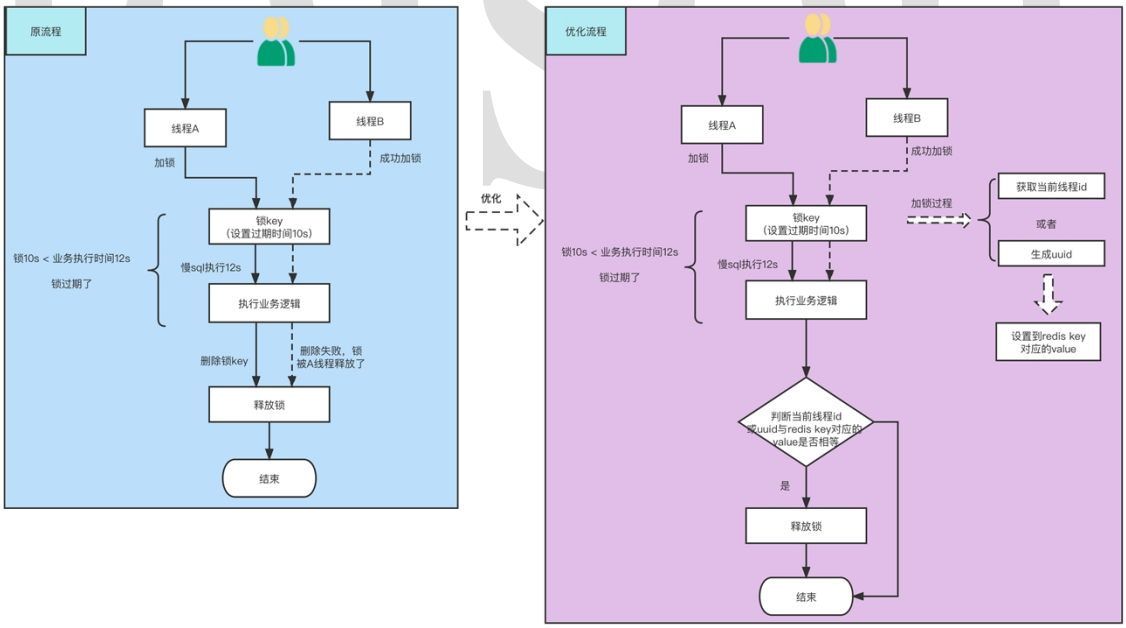
3）如何保证分布式锁不产生死锁？



答：给锁设置一个合理的过期时间，业务执行过程中节点异常宕机，有个兜底终止跳出方案

使用命令：setnx key value ex seconds 设置 key 和对应的过期时间，到了指定的 ex 时间，锁自动释放。

4）如何防止释放别的线程锁？



场景分析：

我们来设想一下这个场景：A、B 两个线程来尝试给 key myLock 加锁，A 线程先拿到锁（假如锁 3 秒后过期），B 线程就在等待尝试获取锁，到这一点毛病没有。

那如果此时业务逻辑比较耗时，执行时间已经超过 redis 锁过期时间，这时 A 线程的锁自动释放（删除 key），B 线程检测到 myLock 这个 key 不存在，执行 SETNX 命令也拿

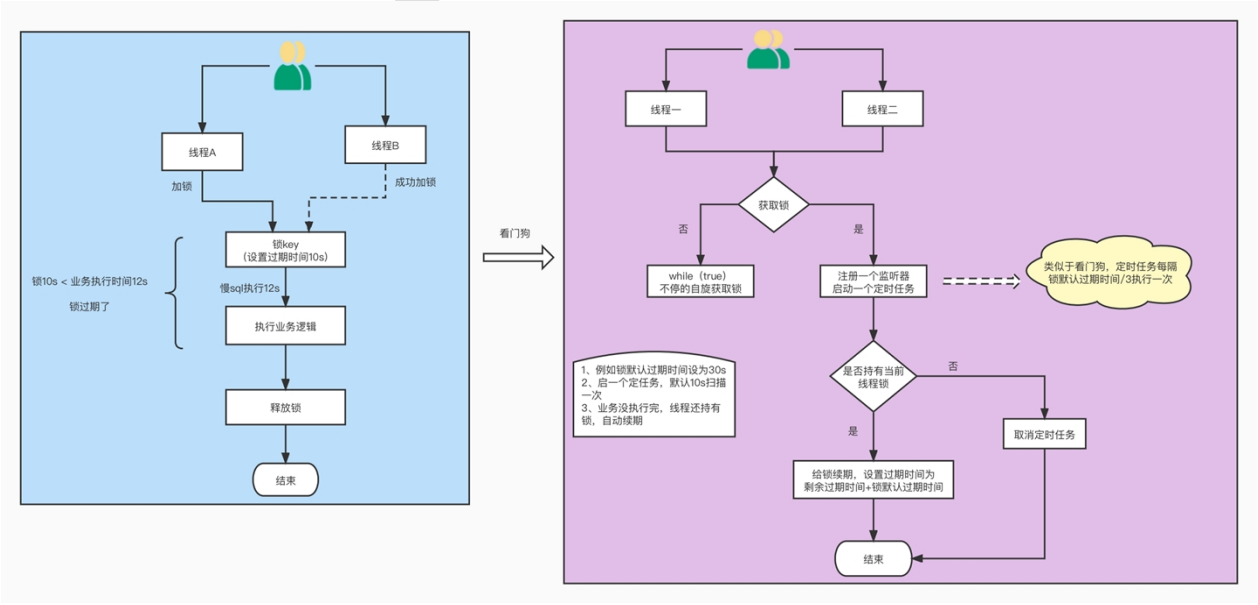
到了锁。但是，此时 A 线程执行完业务逻辑之后，还是会去释放锁（删除 key），这就导致 B 线程的锁被 A 线程给释放了。

答：给锁设置一个当前线程 id 或 uuid 的值，释放锁的时候判断锁的值与当前的值是否相等。

相等才释放锁。以下是伪代码实现：

```
String threadId = Thread.getCurrentThread().getId();   // step1: 生成一个线程 id 或 uuid
setnx key threadId ex seconds       // step2: 加锁，设置 value 为线程 id 或 uuid
if(threadId.equals(redisUtil.get(key)){     //step3:  锁的值与当前的值是否相等
    redisUtil.del(key);       //step4: 释放锁
}
```

5）锁到期了，业务没执行完，如何保证数据的一致？



场景分析：

当线程进来需要的执行时间超过了 Redis key 的过期时间，那么此时锁已经释放了，其他线程就可以立马获得锁，然后执行代码，就又会产生 bug 了。分布式锁 Redis key 的过期时间不管设置成多少都不合适，比如将过期时间设置为 30s，那么如果业务代码出现了类似慢 SQL、查询数据量很大那么过期时间就不管用了。那么这里有没有什么更好的方案呢？

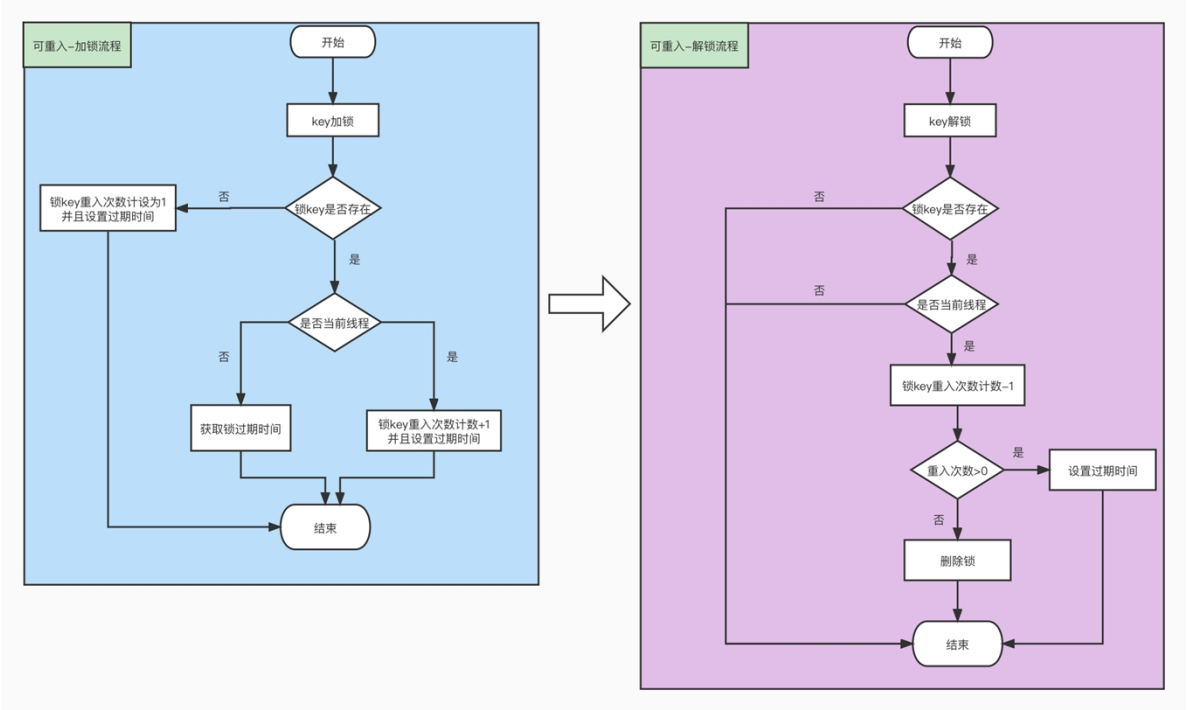
答：常见的处理办法就是采用看门狗机制对分布式锁进行续命，具体步骤如下所示：

当前线程加锁成功后，假设设置默认过期时间为 30 秒，会注册一个定时任务监听这个锁，每隔 30/3=10 秒就去查看这个锁，如果还持有锁，就对锁的过期时间继续续命 30 秒，如果

没持有锁，就取消定时任务。这个机制也被叫做看门狗机制。以下是伪代码实现：

```
ttlRemainingFuture.addListener(new FutureListener<Boolean>() {
        @Override
        public void operationComplete(Future<Boolean> future) throws Exception {
            // 定时任务执行方法,传入当前线程 id，只对当前线程 id 进行续期
            scheduleExpirationRenewal(threadId);
        }
});
private void scheduleExpirationRenewal(final long threadId) {
        // 这里 new 了一个 TimerTask()定时任务器,定时任务会推迟执行，推迟的时间是设置为锁过期时间的 1/3
        int internalLockLeaseTime  =  30;
        Timeout task = commandExecutor.getConnectionManager().newTimeout(new TimerTask() {
            @Override
                public void run(Timeout timeout) throws Exception {
                    //执行业务逻辑
                }
            }, internalLockLeaseTime / 3, TimeUnit.MILLISECONDS);
}
```

6）怎么保证分布式锁可重入？

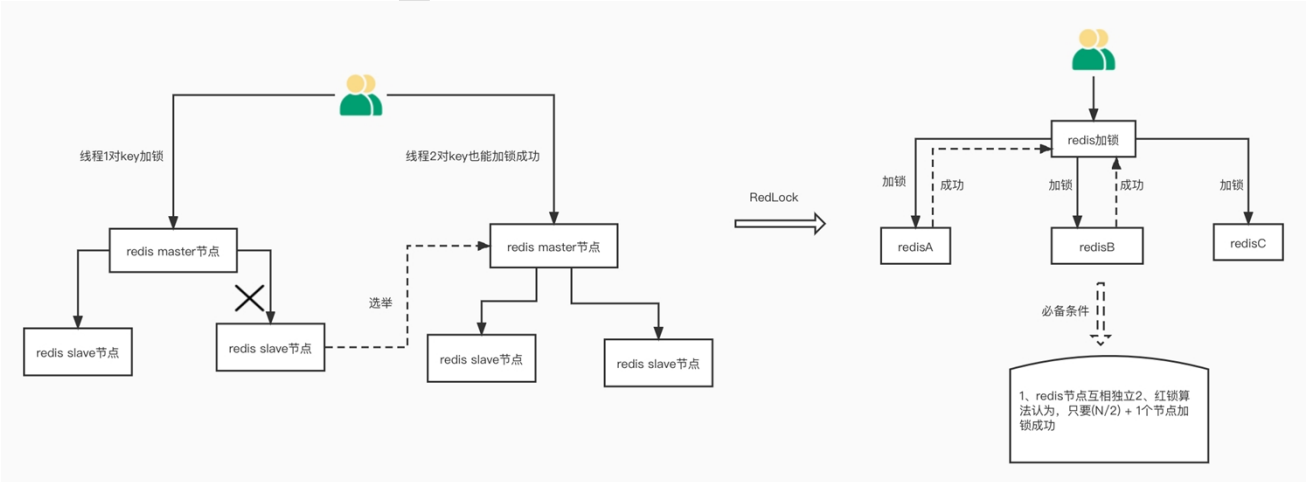


可重入锁的要点是对于同一线程可以多次获取锁，不同线程之间同一把锁不能重复获取。

怎么保证线程可重入获取锁呢？

答：通过维护当前持有锁的计数来实现可重入功能。加锁的时候，第一次获取锁时保存锁的线程标识，后续再次获取锁，先看是否是同一个线程，如果是的话只对锁计数进行递增。解锁时，对锁计数进行递减，同时刷新锁的过期时间。如果计数为 0，最终才释放锁。

7）如何解决 redis 主从节点不同步导致锁失效的问题？



例如：主节点没来的及把刚刚加锁的数据给从节点，主节点就挂了，导致加锁失效。

有些人是不是觉得大佬们都是杠精啊，天天就想着极端情况。其实高可用嘛，拼的就是 99.999...% 中小数点后面的位数。

答：采用红锁算法解决这个锁失效问题，红锁算法认为，只要(N/2) + 1 个节点加锁成功，

那么就认为获取了锁，解锁时将所有实例解锁。

✓ 顺序向 3 个节点请求加锁

✓ 根据一定的超时时间来推断是不是跳过该节点

✓ 2 个节点加锁成功并且花费时间小于锁的有效期

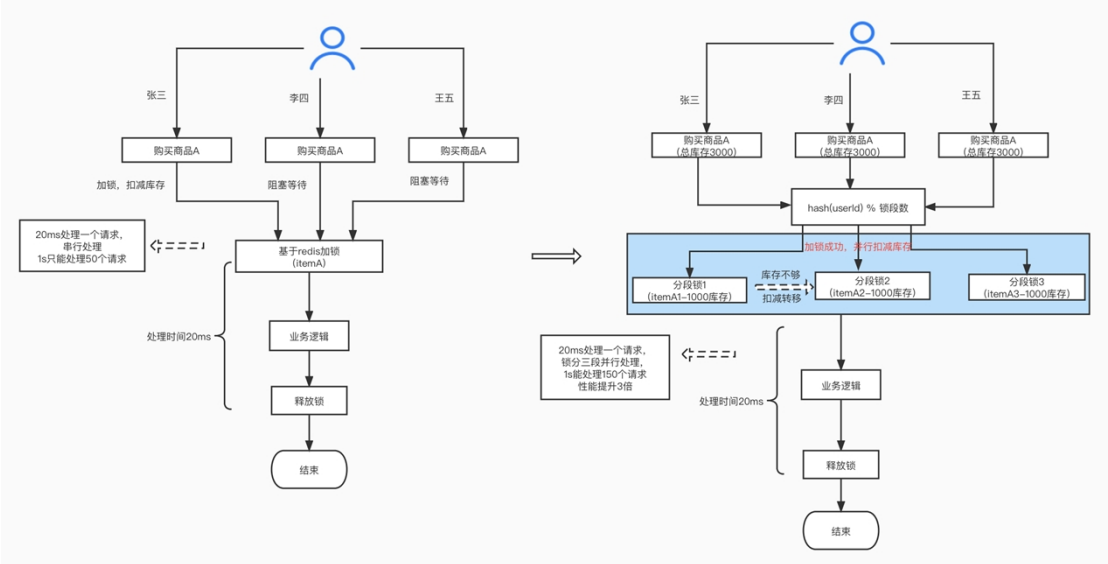
✓ 认定加锁成功

也就是说，假设锁 30 秒过期，2 个节点加锁花了 31 秒，自然是加锁失败了。

这种实现方式，我认为生产上并不推荐使用。很简单原本只需要对一个Redis加锁，设置成功返回即可，但是现在需要对多个 Redis 进行加锁，无形之中增加了好几次网络 IO，万一第一个 Redis 加锁成功后，后面几个 Redis 在加锁过程中出现了类似网络异常的这种情况，那第一个Redis的数据可能就需要做数据回滚操作了，那为了解决一个极低概率发生的问题又引入了多个可能产生的新问题，很显然得不偿失。并且这里还有可能出现更多乱七八糟的问题，所以我认为这种 Redis 分布式锁的实现方式极其不推荐生产使用。

退一万说如果真的需要这种强一致性的分布式锁的话，那为什么不直接用 zk 实现的分布式锁呢，性能肯定也比这个 RedLock 的性能要好.

8）如何优化高并发下锁性能？



答：与 ConcurrentHashMap 的设计思想有点类似，用分段锁来实现。

假如 A 商品的库存是 3000 个，现在可以将该 A 商品的 3000 个库存利用类似 ConcurrentHashMap 的原理将不同数量段位的库存的利用取模或者是 hash 算法让其扩容到不同的节点上去，这样这 3000 的库存就水平扩容到了多个 Redis 节点上，然后请求 Redis 拿库存的时候请求原本只能从一个 Redis 上取数据，现在可以从 3 个 Redis 上取数据，从

而可以大大提高并发效率。

五、不同场景如何选择分布式锁实现方案

1）redis 分布式锁（推荐）

互联网项目并发量高，对性能要求高，比较推荐。

redis 常见操作，例如基本类型 string、hash、list、set 等等操作可以采用 jedis 或 lettuce。

对于跟分布式锁相关的操作集成 redission。

2）分布式锁百分百可靠

可以选用 Zookeeper 作为分布式锁。采用 cap 理论中的 cp 模型保证高可靠性。

一般的项目我们可以结合不同的场景，同时兼容两种分布锁的实现。