

《我想进大厂》之分布式锁夺命连环9问 | 大理版人在囧途

原创 艾小仙 艾小仙 3月8日

收录于话题

#面试大全 29 #文章精选汇总 78

开个头，这是篇技术文章，但是昨天一天太恶心了，忍不住还是简单说下昨天的事情。

昨天早上11点飞大理，结果9点钟要出门的时候发现密码锁坏了，不用密码都能打开，一边司机师傅在催着走，一边连忙打电话给房东和客服找人维修，这是第一。

然后飞机晚点，11点20飞到4点钟才要落地，下降的过程那叫一个颠簸，我以为都要没了，这也是第一次晕飞机，简直快吐了，这是第二。



然后快4点了，飞机总算快要降落了，轮子都快着地了，结果愣是拔起来又起飞了，最后知道是大理8级大风，机长不敢落地。。。这是第三。

最后通知起飞不知道什么时候，要等大理那边通知，没有办法，我们只好下飞机转高铁，急急忙忙的一路转，总算赶上了最后7点前的高铁，否则就要等到9点以后了，最后一路周转，9点多总算到了酒店，好在酒店还算行，没有让我太过于失望。

这一天搞下来，整个一人在囧途，太累了。好吧，废话就这么多，文章开始。

说说分布式锁吧？

对于一个单机的系统，我们可以通过synchronized或者ReentrantLock等这些常规的加锁方式来实现，然而对于一个分布式集群的系统而言，单纯的本地锁已经无法解决问题，所以需要用到分布式锁了，通常我们都会引入三方组件或者服务来解决这个问题，比如数据库、Redis、Zookeeper等。

通常来说，分布式锁要保证互斥性、不死锁、可重入等特点。

互斥性指的是对于同一个资源，任意时刻，都只有一个客户端能持有锁。

不死锁指的是必须要有锁超时这种机制，保证在出现问题的时候释放锁，不会出现死锁的问题。

可重入指的是对于同一个线程，可以多次重复加锁。

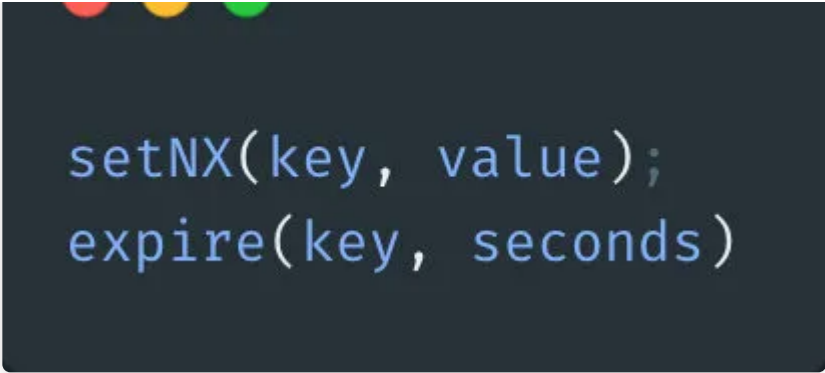
那你分别说说使用数据库、Redis和Zookeeper的实现原理？

数据库的话可以使用乐观锁或者悲观锁的实现方式。

乐观锁通常就是数据库中我们会有一个版本号，更新数据的时候通过版本号来更新，这样的话效率会比较高，悲观锁则是通过 `for update` 的方式，但是会带来很多问题，因为他是一个行级锁，高并发的情况下可能会导致死锁、客户端连接超时等问题，一般不推荐使用这种方式。

Redis是通过 `set` 命令来实现，在 2.6.2 版本之前，实现方式可能是这样：





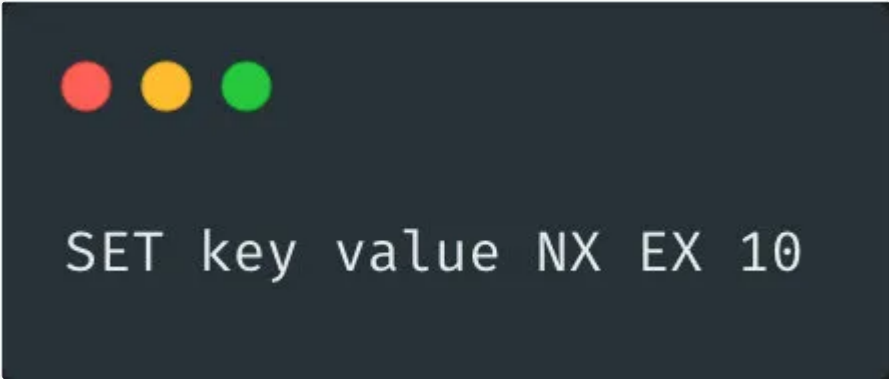
```
setNX(key, value);  
expire(key, seconds)
```

`setNX` 命令代表当 `key` 不存在时返回成功，否则返回失败。

但是这种实现方式把加锁和设置过期时间的步骤分成两步，他们并不是原子操作，如果加锁成功之后程序崩溃、服务宕机等异常情况，导致没有设置过期时间，那么就会导致死锁的问题，其他线程永远都无法获取这个锁。

之后的版本中，Redis提供了原生的 `set` 命令，相当于两命令合二为一，不存在原子性的问题，当然也可以通过lua脚本来解决。

`set` 命令如下格式：



```
SET key value NX EX 10
```

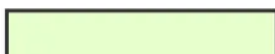
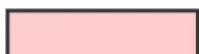
`key` 为分布式锁的key

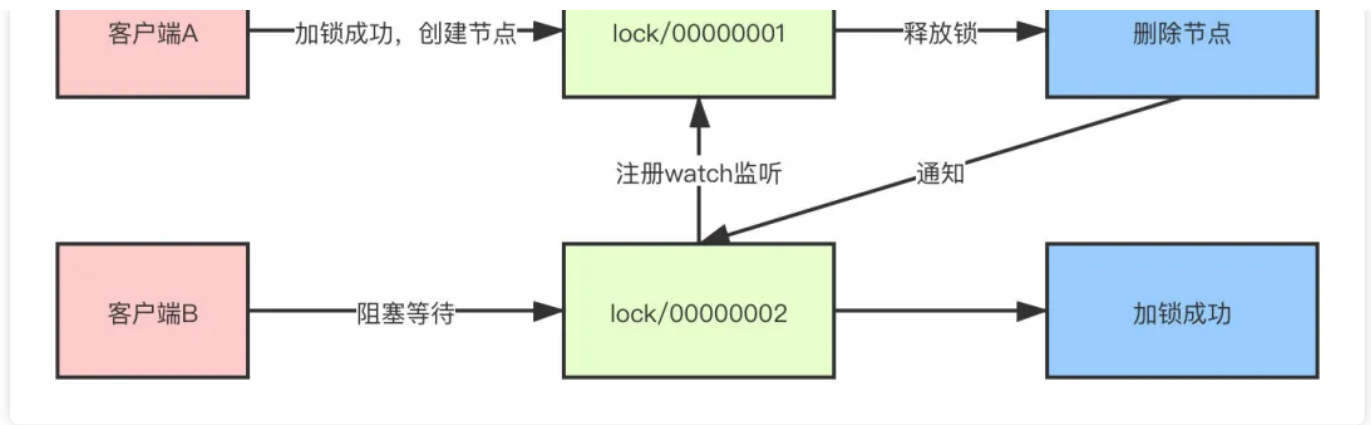
`value` 为分布式锁的值，一般为不同的客户端设置不同的值

`NX` 代表如果要设置的key已存在，则取消设置

`EX` 代表过期时间为秒，`PX`则为毫秒，比如上面示例中为10秒过期

Zookeeper是通过创建临时顺序节点的方式来实现。





1. 当需要对资源进行加锁时，实际上就是在父节点之下创建一个临时顺序节点。
2. 客户端A来对资源加锁，首先判断当前创建的节点是否为最小节点，如果是，那么加锁成功，后续加锁线程阻塞等待
3. 此时，客户端B也来尝试加锁，由于客户端A已经加锁成功，所以客户端B发现自己的节点并不是最小节点，就会去取到上一个节点，并且对上一节点注册监听
4. 当客户端A操作完成，释放锁的操作就是删除这个节点，这样就可以触发监听事件，客户端B就会得到通知，同样，客户端B判断自己是否为最小节点，如果是，那么则加锁成功

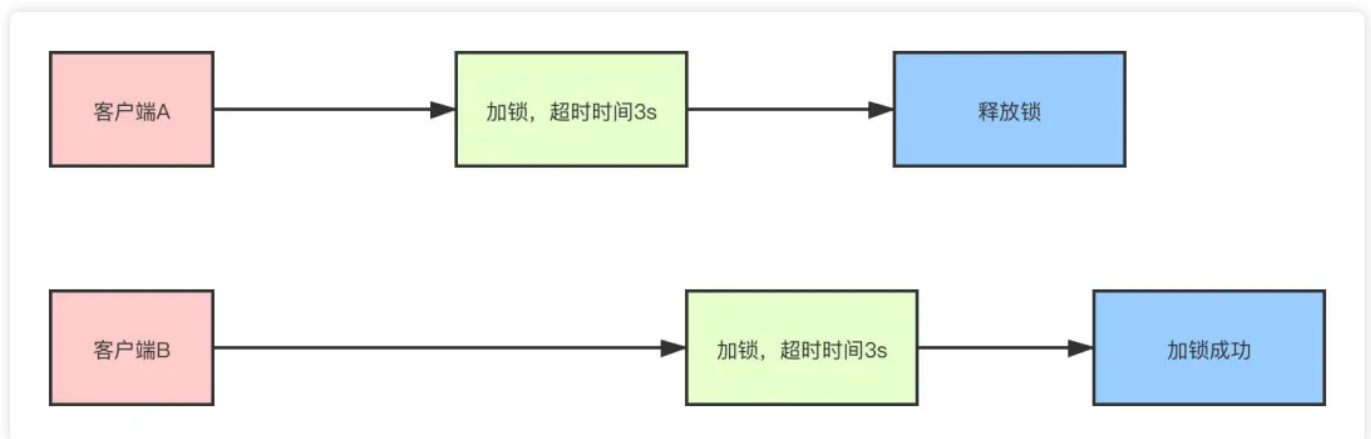
你说改为set命令之后就解决了问题？那么还会有其他的问题呢？

虽然 `set` 解决了原子性的问题，但是还是会存在两个问题。

锁超时问题

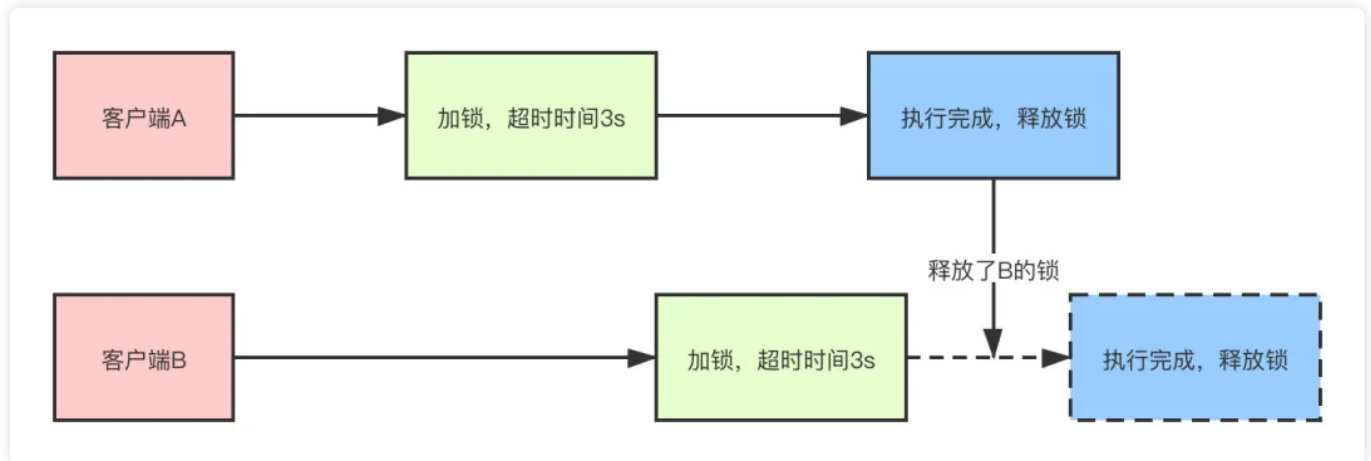
比如客户端A加锁同时设置超时时间是3秒，结果3s之后程序逻辑还没有执行完成，锁已经释放。客户端B此时也来尝试加锁，那么客户端B也会加锁成功。

这样的话，就导致了并发的的问题，如果代码幂等性没有处理好，就会导致问题产生。



锁误删除

还是类似的问题，客户端A加锁同时设置超时时间3秒，结果3s之后程序逻辑还没有执行完成，锁已经释放。客户端B此时也来尝试加锁，这时客户端A代码执行完成，执行释放锁，结果释放了客户端B的锁。



那上面两个问题你有什么好的解决方案吗？

锁超时

这个有两个解决方案。

1. 针对锁超时的问题，我们可以根据平时业务执行时间做大致的评估，然后根据评估的时间设置一个较为合理的超时时间，这样能一大部分程度上避免问题。
2. 自动续租，通过其他的线程为将要过期的锁延长持有时间

锁误删除

每个客户端的锁只能自己解锁，一般我们可以在使用 `set` 命令的时候生成随机的 `value`，解锁使用lua脚本判断当前锁是否自己持有的，是自己的锁才能释放。

```
#加锁
SET key random_value NX EX 10

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

了解RedLock算法吗？

因为在Redis的主从架构下，主从同步是异步的，如果在Master节点加锁成功后，指令还没有同步到Slave节点，此时Master挂掉，Slave被提升为Master，新的Master上并没有锁的数据，其他的客户端仍然可以加锁成功。

对于这种问题，Redis作者提出了RedLock红锁的概念。

RedLock的理念下需要至少2个Master节点，多个Master节点之间完全互相独立，彼此之间不存在主从同步和数据复制。

主要步骤如下：

1. 获取当前Unix时间
2. 按照顺序依次尝试从多个节点锁，如果获取锁的时间小于超时时间，并且超过半数的节点获取成功，那么加锁成功。这样做的目的就是为了避免某些节点已经宕机的情况下，客户端还在一直等待响应结果。举个例子，假设现在有5个节点，过期时间=100ms，第一个节点获取锁花费10ms，第二个节点花费20ms，第三个节点花费30ms，那么最后锁的过期时间就是100-(10+20+30)，这样就是加锁成功，反之如果最后时间<0，那么加锁失败
3. 如果加锁失败，那么要释放所有节点上的锁

那么RedLock有什么问题吗？

其实RedLock存在不少问题，所以现在其实一般不推荐使用这种方式，而是推荐使用Redisson的方案，他的问题主要如下几点。

性能、资源

因为需要对多个节点分别加锁和解锁，而一般分布式锁的应用场景都是在高并发的情况下，所以耗时较长，对性能有一定的影响。此外因为需要多个节点，使用的资源也比较多，简单来说就是费钱。

节点崩溃重启

比如有1~5号五个节点，并且没有开启持久化，客户端A在1，2，3号节点加锁成功，此时3号节点崩溃宕机后发生重启，就丢失了加锁信息，客户端B在3，4，5号节点加锁

成功。

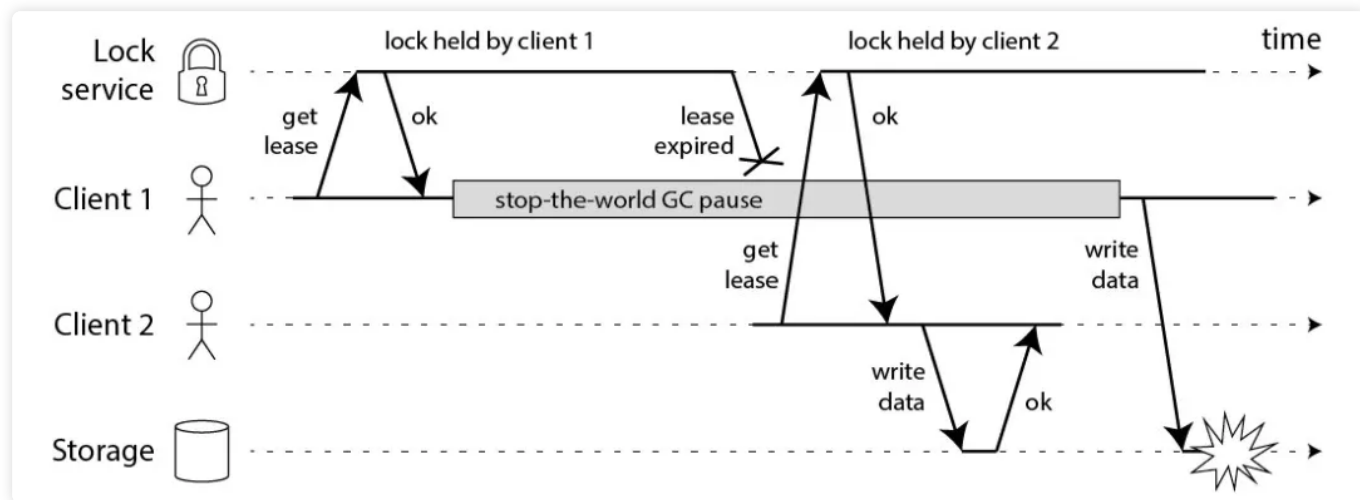
那么，两个客户端A\B同时获取到了同一个锁，问题产生了，怎么解决？

1. Redis作者建议的方式就是延时重启，比如3号节点宕机之后不要立刻重启，而是等待一段时间后再重启，这个时间必须大于锁的有效时间，也就是锁失效后再重启，这种人为干预的措施真正实施起来就比较困难了
2. 第二个方案那么就是开启持久化，但是这样对性能又造成了影响。比如如果开启AOF默认每秒一次刷盘，那么最多丢失一秒的数据，如果想完全不丢失的话就对性能造成较大的影响。

GC、网络延迟

对于RedLock，Martin Kleppmann提出了很多质疑，我就只举这样一个GC或者网络导致的例子。（这个问题比较多，我就不一一举例了，心里有一个概念就行了，文章地址：<https://martin.kleppmann.com/2016/02/08/how-to-do-distributed-locking.html>）

从图中我们可以看出，client1先获取到锁，然后发生GC停顿，超过了锁的有效时间导致锁被释放，然后锁被client2拿到，然后两个客户端同时拿到锁在写数据，问题产生。



图片来自Martin Kleppmann

时钟跳跃

同样的例子，假设发生网络分区，4、5号节点变为一个独立的子网，3号节点发生始终跳跃（不管人为操作还是同步导致）导致锁过期，这时候另外的客户端就可以从3、4、5号节点加锁成功，问题又发生了。

那你说有什么好的解决方案吗？

上面也提到了，其实比较好的方式是使用 `Redission`，它是一个开源的Java版本的Redis客户端，无论单机、哨兵、集群环境都能支持，另外还很好地解决了锁超时、公平不公平锁、可重入等问题，也实现了 `RedLock`，同时也是官方推荐的客户端版本。

那么Redission实现原理呢？

加锁、可重入

首先，加锁和解锁都是通过lua脚本去实现的，这样做的好处是为了兼容老版本的redis同时保证原子性。

`KEYS[1]` 为锁的key，`ARGV[2]` 为锁的value，格式为uuid+线程ID，`ARGV[1]` 为过期时间。

主要的加锁逻辑也比较容易看懂，如果 `key` 不存在，通过hash的方式保存，同时设置过期时间，反之如果存在就是+1。

对应的就是 `hincrby', KEYS[1], ARGV[2], 1` 这段命令，对hash结构的锁重入次数+1。

```
<T> RFuture<T> tryLockInnerAsync(long leaseTime, TimeUnit unit, long threadId, RedisStrictCommand<T> command) {
    internalLockLeaseTime = unit.toMillis(leaseTime);

    return commandExecutor.evalWriteAsync(getName(), LongCodec.INSTANCE, command,
        "if (redis.call('exists', KEYS[1]) = 0) then " +
            "redis.call('hset', KEYS[1], ARGV[2], 1); " +
            "redis.call('pexpire', KEYS[1], ARGV[1]); " +
            "return nil; " +
        "end; " +
        "if (redis.call('hexists', KEYS[1], ARGV[2]) = 1) then " +
            "redis.call('hincrby', KEYS[1], ARGV[2], 1); " +
            "redis.call('pexpire', KEYS[1], ARGV[1]); " +
            "return nil; " +
        "end; " +
        "return redis.call('pttl', KEYS[1]);",
        Collections.<Object>singletonList(getName()), internalLockLeaseTime, getLockName(threadId));
}

protected String getLockName(long threadId) {
    return id + ":" + threadId;
}
```

解锁

1. 如果key都不存在了，那么就返回
2. 如果key、field不匹配，那么说明不是自己的锁，不能释放，返回空
3. 释放锁，重入次数-1，如果还大于0那么久刷新过期时间，反之那么久删除锁

```
public RFuture<Void> unlockAsync() {
    long threadId = Thread.currentThread().getId();
    return unlockAsync(threadId);
}

protected RFuture<Boolean> unlockInnerAsync(long threadId) {
    return commandExecutor.evalWriteAsync(getName(), LongCodec.INSTANCE, RedisCommands.EVAL_BOOLEAN,
        "if (redis.call('exists', KEYS[1]) = 0) then " +
        "    redis.call('publish', KEYS[2], ARGV[1]); " +
        "    return 1; " +
        "end; " +
        "if (redis.call('hexists', KEYS[1], ARGV[3]) = 0) then " +
        "    return nil; " +
        "end; " +
        "local counter = redis.call('hincrby', KEYS[1], ARGV[3], -1); " +
        "if (counter > 0) then " +
        "    redis.call('pexpire', KEYS[1], ARGV[2]); " +
        "    return 0; " +
        "else " +
        "    redis.call('del', KEYS[1]); " +
        "    redis.call('publish', KEYS[2], ARGV[1]); " +
        "    return 1; " +
        "end; " +
        "return nil;",
        Arrays.<Object>asList(getName(), getChannelName(), LockPubSub.unlockMessage, internalLockLeaseTime,
            getLockName(threadId));
    }

    protected String getLockName(long threadId) {
        return id + ":" + threadId;
    }
}
```

watchdog

也叫做看门狗，也就是解决了锁超时导致的问题，实际上就是一个后台线程，默认每隔10秒自动延长锁的过期时间。

默认的时间就是 `internalLockLeaseTime / 3`，`internalLockLeaseTime` 默认为30秒。

```
private void scheduleExpirationRenewal(final long threadId) {
    if (expirationRenewalMap.containsKey(getEntryName())) {
        return;
    }

    Timeout task = commandExecutor.getConnectionManager().newTimeout(new TimerTask() {
        @Override
        public void run(Timeout timeout) throws Exception {
            // ...
        }
    }, internalLockLeaseTime / 3, TimeUnit.SECONDS);
}
```

```

        RFuture<Boolean> future = renewExpirationAsync(threadId);

        future.addListener(new FutureListener<Boolean>() {
            @Override
            public void operationComplete(Future<Boolean> future) throws Exception {
                expirationRenewalMap.remove(getEntryName());
                if (!future.isSuccess()) {
                    log.error("Can't update lock " + getName() + " expiration", future.cause());
                    return;
                }

                if (future.getNow()) {
                    // reschedule itself
                    scheduleExpirationRenewal(threadId);
                }
            }
        });
    }

    }, internalLockLeaseTime / 3, TimeUnit.MILLISECONDS);

    if (expirationRenewalMap.putIfAbsent(getEntryName(), new ExpirationEntry(threadId, task)) != null) {
        task.cancel();
    }
}

```

最后，实际生产中对于不同的场景该如何选择？

首先，如果对于并发不高并且比较简单的场景，通过数据库乐观锁或者唯一主键的形式就能解决大部分的问题。

然后，对于Redis实现的分布式锁来说性能高，自己去实现的话比较麻烦，要解决锁续租、lua脚本、可重入等一系列复杂的问题。

对于单机模式而言，存在单点问题。

对于主从架构或者哨兵模式，故障转移会发生锁丢失的问题，因此产生了红锁，但是红锁的问题也比较多，并不推荐使用，推荐的使用方式是用Redisson。

但是，不管选择哪种方式，本身对于Redis来说不是强一致性的，某些极端场景下还是可能会存在问题。

对于Zookeeper的实现方式而言，本身就是保证数据一致性的，可靠性更高，所以不存在Redis的各种故障转移带来的问题，自己实现也比较简单，但是性能相比Redis稍差。

不过，实际中我们当然是有啥用啥，老板说用什么就用什么，我才不管那么多。



艾小仙

一个愤世嫉俗，脱离低级趣味的人

86篇原创内容

公众号



.....END.....

【 往期回顾 】

关于MVCC，我之前写错了，这次我改好了！

长篇连载(一)你的编程能力从什么时候开始突飞猛进？

好久没更新，新年第一篇！

好了，我摊牌了，B站，牛逼！

长篇连载，人生30年(二)：职场菜鸟被开除

《我想进大厂》之Zookeeper夺命连环9问

收录于话题 #面试大全·29个

上一篇

阿里二面：什么是mmap？

下一篇

真实字节二面：什么是伪共享？

文章已于2021/03/09修改

喜欢此内容的人还喜欢

我用kafka两年踩过的一些非比寻常的坑

苏三说技术

缓存最关心哪些指标？

冰河技术

为啥有的人一晒就黑，有的人却越晒越白？

