Redis在京东到家的订单中的使用

发表于 2017-06-30 | 分类于 架构

背景

Redis作为一款性能优异的内存数据库，在互联网公司有着多种应用场景，下面介绍下Redis在京东到家的订单列表中的使用场景。主要从以下几个方面来介绍：

1. 订单列表在Redis中的存储结构
2. Redis和DB数据一致性保证
3. Redis中的分布式锁
4. 缓存防穿透和雪崩

订单列表在Redis中的存储结构

- 订单列表数据在缓存中，是以用户的唯一标识作为键，以一个按下单时间倒序的有序集合为值进行存储的。大家都知道Redis的sorted set中每个元素都有一个分数，Redis就是根据这个分数排序的。订单有序集合中的每个元素是将时间毫秒数+订单号最后3位作为分数进行排序的。为什么不只用毫秒数作为分数呢？因为我们的下单时间只精确到秒，如果不加订单号最后3位，若同一秒有两个或两个以上订单时，排序分数就会一样，从而导致根据分数从缓存查询订单时不能保证唯一性。而我们的订单号的生成规则可以保证同一秒内的订单号的最后3位肯定不一样，从而可以解决上述问题。
- 有必要将一个用户的所有订单都放入缓存吗？针对用户订单是没有必要的，因为很少有用户去看很久以前的历史订单。真正的热点数据其实也就是最近下过的一些订单，所以，为了节省内存空间，只需要存放一个用户最近下过的N条订单就行了，这个N，相当于一个阀值，超过了这个阀值，再从数据库中查询订单数据，当然，这部分查库操作已经是很小概率的操作了。

Redis和DB数据一致性保证

只要有多份数据，就会涉及到数据一致性的问题。Redis和数据库的数据一致性，也是必然要面对的问题。我们这边的订单数据是先更新数据库，数据库更新成功后，再更新缓存，若数据库操作成功，缓存操作失败了，就出现了数据不一致的情况。保证数据一致性我们前后使用过两种方式：

- 方式一：

1. 循环5次更新缓存操作，直到更新成功退出循环，这一步主要能减小由于网络瞬间抖动导致的更新缓存失败的概率，对于缓存接口长时间不可用，靠循环调用更新接口是不能补救接口调用失败的。
2. 如果循环5次还没有更新成功，就通过worker去定时扫描出数据库的数据，去和缓存中的数据进行比较，对缓存中的状态不正确的数据进行纠正。

- 方式二：

1. 跟方式一的第一步操作一样
2. 若循环更新5次仍不成功，则发一个缓存更新失败的mq，通过消费mq去更新缓存，会比通过定时任务扫描更及时，也不会有扫库的耗时操作。此方式也是我们现在使用的方式。

   代码示例：

```
for (int i = 0; i < 5; i++) {
    try {
        // 入缓存操作
        addOrderListRedis(key, score, orderListVO);
        break;
    } catch (Exception e) {
        log.error("{}IOrderRedisCache.putOrderList2OrderListRedis--->>jdCacheCloud.zAdd exception: ", logSid, e
```

```
        if (i == 4) sendUpOrderCacheMQ(orderListVO, logSid); // 如果循环5次，仍添加缓存失败，发送MQ，通过MQ继续更新缓存
    }

```

Redis中的分布式锁

分布式锁常用的实现方式有Redis和zookeeper，本文主要介绍下Redis的分布式锁，然后再介绍下我们使用分布式锁的场景。

Redis分布式锁在2.6.12版本之后的实现方式比较简单，只需要使用一个命令即可：

```
SET key value [EX seconds] [NX]
```

其中，可选参数EX seconds ： 设置键的过期时间为 seconds 秒；NX ： 只在键不存在时，才对键进行设置操作。

这个命令相当于2.6.12之前的setNx和expire两个命令的原子操作命令。Redis的JAVA客户端分布式锁实现示例代码：

2.6.12版本之后：

```
public booelan getLock(String lockKey, String lockValue){
  if(shardedXCommands.set(key,lockValue,10,TimeUnit.SECONDS,false)) {
      return true;
  }
  return false;
}
```

2.6.12版本之前，由于没有一个上述的原子命令，需要一些命令组合实现，但不能简单的使用setNx、expire这两个命令，因为如果setNx成功，expire命令失败时，恰好执行删除lockKey的也执行失败，key就永远不会过期，就会出现死锁问题，如：

```
public booelan getLock(String lockKey, String lockValue) {
  boolean lock = false;
  try{
      lock = shardedXCommands.setNX(lockKey, lockValue);
      shardedXCommands.expire(lockKey, 5); // (1) 这个命令执行失败
  } catch(Exception e) {
  }

  return lock;
}
public void A(){
  try{
      if(getLock("订单号", "lockValue")) {
          // doSomething
      }
  } finally {
      unLock("订单号");// (2)系统崩溃，解锁失败
  }
}
```

第(1)步设置lockKey失效时间失败，lockKey在缓存永久保存。

第(2)步没来得及释放锁时，系统崩溃，finally块没来得及执行，最终导致锁永远在缓存中，所有其他线程再也获取不到锁。所以不能单纯的依靠设置锁的失效时间来防止释放锁失败，需要通过下列方法防止这种情况，但比较繁琐，不过2.6.12版本之前也必须通过如下方法才更为妥当：

```
public booelan getLock(String lockKey) {
  boolean lock = false;
  while (!lock) {
```

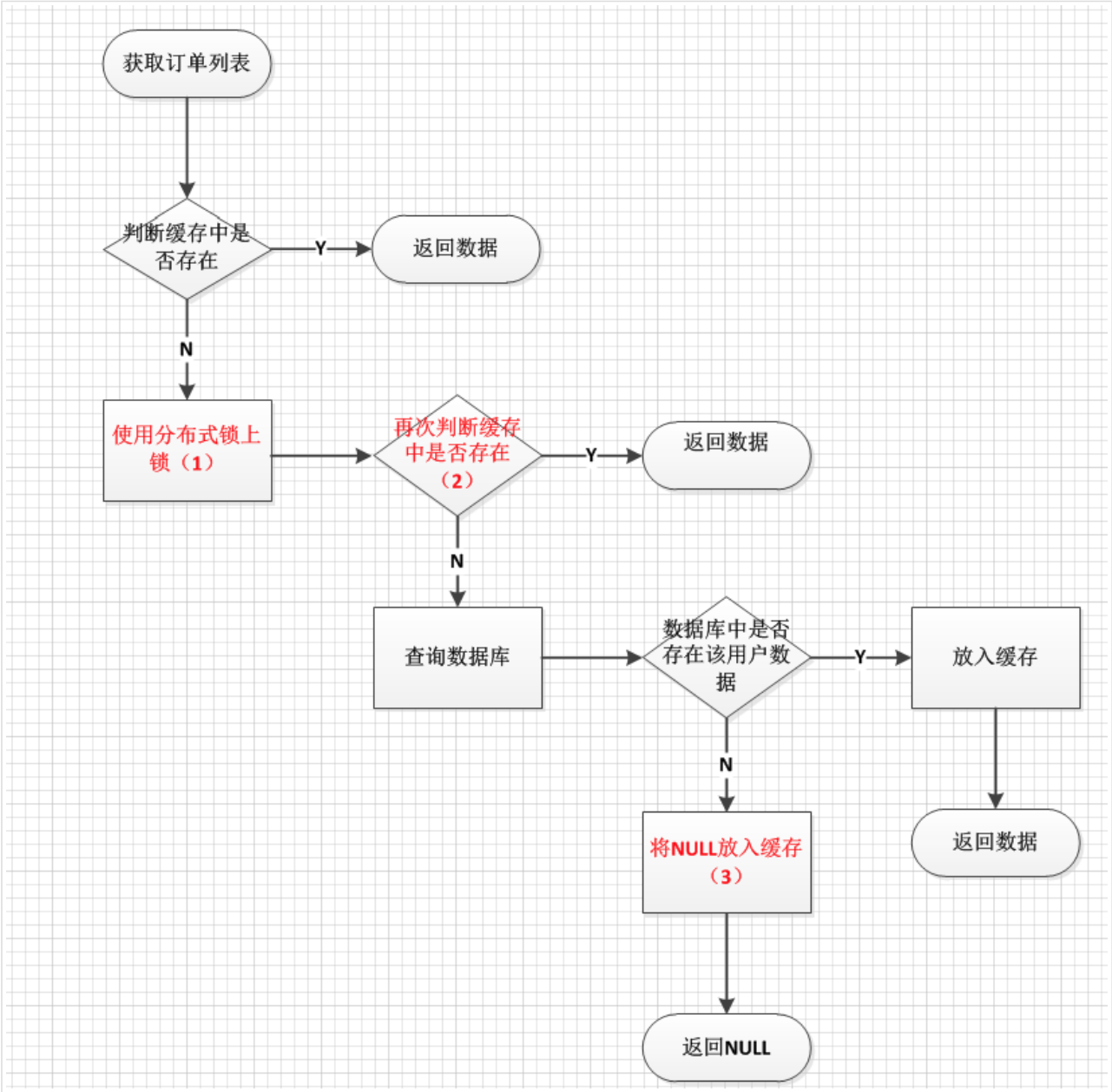
```
        String expireTime = String.valueOf(System.currentTimeMillis() + 5000);
        // (1)第一个获得锁的线程，将lockKey的值设置为当前时间+5000毫秒，后面会判断，如果5秒之后，获得锁的线程还没有执行完，会忽略之前获
        lock = shardedXCommands.setNX(lockKey, expireTime);
        if (lock) { // 已经获取了这个锁 直接返回已经获得锁的标识
            return lock;
        }
         // 没获得锁的线程可以执行到这里：从Redis获取老的时间戳
        String oldTimeStr = shardedXCommands.get(lockKey);
        if (oldTimeStr != null && !"".equals(oldTimeStr.trim())) {
            Long oldTimeLong = Long.valueOf(oldTimeStr);
            // 当前的时间戳
            Long currentTimeLong = System.currentTimeMillis();
            // (2)如果oldTimeLong小于当前时间了，说明之前持有锁的线程执行时间大于5秒了，就强制忽略该线程所持有的锁，重新设置自己的锁
            if (oldTimeLong < currentTimeLong) {
                // (3)调用getset方法获取之前的时间戳,注意这里会出现多个线程竞争，但肯定只会有一个线程会拿到第一次获取到锁时设置的expi
                String oldTimeStr2 = shardedXCommands.getSet(lockKey, String.valueOf(System.currentTimeMillis() + 50
                // (4)如果刚获取的时间戳和之前获取的时间戳一样的话,说明没有其他线程在占用这个锁,则此线程可以获取这个锁.
                if (oldTimeStr2 != null && oldTimeStr.equals(oldTimeStr2)) {
                    lock = true; // 获取锁标记
                    break;
                }
            }
        }
        // 暂停50ms,重新循环
        try {
            Thread.sleep(50);
        } catch (InterruptedException e) {
            log.error(e);
        }
    }
    return lock;
}
```

上述方法主要使用了Redis的setNX、getSet两个方法，不依赖Redis的expire方法，即便是删除锁失败时，上面逻辑第（2）步也会规避这个问题。

- 订单使用分布式锁的场景是订单状态有变更的时候，需要先使用锁—>读缓存数据—>判断当前订单状态是否允许变更为别的状态—>更新缓存中的订单状态—>释放锁。

缓存防穿透和雪崩

- 缓存为我们挡住了80-90%甚至更多的流量，然而当缓存中的大量热点数据恰巧在差不多的时间过期时，或者当有人恶意伪造一些缓存中根本没有的数据疯狂刷接口时，就会有大量的请求直接穿透缓存访问到数据库（因为查询数据策略是缓存没有命中，就查数据库），给数据库造成巨大压力，甚至使数据库崩溃，这肯定是我们系统不允许出现的情况。我们需要针对这种情况进行处理。下图是处理流程图：



代码示例：

```
// 代码段1
// 锁的数量 锁的数量越少 每个用户对锁的竞争就越激烈，直接打到数据库的流量就越少，对数据库的保护就越好，如果太小，又会影响系统吞吐量，
public static final String[] LOCKS = new String[128];
// 在静态块中将128个锁先初始化出来
static {
   for (int i = 0; i < 128; i++) {
       LOCKS[i] = "lock_" + i;
   }
}

// 代码段2
public List<OrderVOList> getOrderVOList(String userId) {
   List<OrderVOList> list = null;
   // 1.先判断缓存中是否有这个用户的数据，有就直接从缓存中查询并返回
   if (orderRedisCache.isOrderListExist(userId)) {
       return  getOrderListFromCache(userId);
   }
   // 2.缓存中没有，就先上锁，锁的粒度是根据用户Id的hashcode和127取模
   String[] locks = OrderRedisKey.LOCKS;
   int index = userId.hashCode() & (locks.length - 1);
   try {
       // 3.此处加锁很有必要，加锁会保证获取同一个用户数据的所有线程中，只有一个线程可以访问数据库，从而起到减小数据库压力的作用
       orderRedisCache.lock(locks[index]);
```

```
        // 4.上锁之后再判断缓存是否存在，为了防止再获得锁之前，已经有别的线程将数据加载到缓存，就不允许再查询数据库了。
        if (orderRedisCache.isOrderListExist(userId)) {
            return getOrderListFromCache(userId);
        }
        // 查询数据库
        list = getOrderListFromDb(userId);
        // 如果数据库没有查询出来数据，则在缓存中放入NULL，标识这个用户真的没有数据，等有新订单入库时，会删掉此标识，并放入订单数据
        if(list == null || list.size() == 0) {
            jdCacheCloud.zAdd(OrderRedisKey.getListKey(userId), 0, null);
        } else {
            jdCacheCloud.zAdd(OrderRedisKey.getListKey(userId), list);
        }
        return list;
    } finally {
        orderRedisCache.unlock(locks[index]);
    }

```

防止穿透和雪崩的关键地方在于使用分布式锁和锁的粒度控制。首先初始化了128（0–127）个锁，然后让所有缓存没命中的用户去竞争这128个锁，得到锁后并且再一次判断缓存中依然没有数据的，才有权利去查询数据库。没有将锁粒度限制到用户级别，是因为如果粒度太小的话，某一个时间点有太多的用户去请求，同样会有很多的请求打到数据库。比如：
在时间点T1有10000个用户的缓存数据失效了，恰恰他们又在时间点T1都请求数据，如果锁粒度是用户级别，那么这10000个用户都会有各自的锁，也就意味着他们都可以去访问数据库，同样会对数据库造成巨大压力。而如果是通过用户id去hashcode和127取模，意味着最多会产生128个锁，最多会有128个并发请求访问到数据库，其他的请求会由于没有竞争到锁而阻塞，待第一批获取到锁的线程释放锁之后，剩下的请求再进行竞争锁，但此次竞争到锁的线程，在执行代码段2中第4步时：orderRedisCache.isOrderListExist(userId)，缓存中有可能已经有数据了，就不用再查数据库了，依次类推，从而可以挡住很多数据库请求，起到很好的保护数据库的作用。

总结

1. 缓存中存放了用户的部分订单，且是以下单时间+订单号最后三位算出分数（这样做是为因为下单时间只精确到秒，为了防止同一秒下多个订单导致排序分数相同），进行排序的有序集合。
2. 数据库更新成功，缓存更新失败，这样导致数据不一致，可以通过更新缓存失败后发mq的策略进行缓存更新尝试，比定时任务更高效，更及时。
3. Redis分布式锁实现，2.6版本前，通过setNx和getSet两个命令实现，2.6版本之后，Redis提供了SET key value [EX seconds] [NX]这个命令可以实现。
4. 防穿透和雪崩依赖了分布式锁，值得注意的是锁粒度不能细到用户级别，可以根据数据库性能和业务要求，算出合适的锁的数量，让所有未命中缓存的用户通过hashCode和锁数量取模，去竞争锁，得到锁的才获得查库权利。

#Redis

‹ JVM安全退出