分布式锁常用方案

王波15041992

分布式锁解决方案：

1. 首先明确一点，有人可能会问是否可以考虑采用ReentrantLock来实现，但是实际上去实现的时候是有问题的，ReentrantLock的lock和unlock要求必须是在同一线程进行，而分布式应用中，lock和unlock是两次不相关的请求，因此肯定不是同一线程，因此导致无法使用ReentrantLock。

2. 基于数据库表做乐观锁，用于分布式锁。

3. 使用memcached的add()方法，用于分布式锁。

4. 使用memcached的cas()方法，用于分布式锁。(不常用)

5. 使用redis的setnx()、expire()方法，用于分布式锁。

6. 使用redis的setnx()、get()、getset()方法，用于分布式锁。

7. 使用redis的watch、multi、exec命令，用于分布式锁。(不常用)

8. 使用zookeeper，用于分布式锁。(不常用)

基于数据库资源表做乐观锁，用于分布式锁:

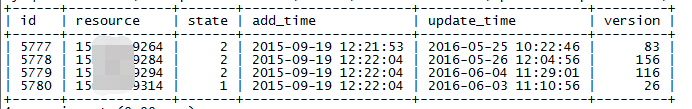
1. 首先说明**乐观锁的含义:**

          大多数是基于数据版本(version)的记录机制实现的。何谓数据版本号？即为数据增加一个版本标识，在基于数据库表的版本解决方案中，一般是通过为数据库表添加一个 “version”字段来实现读取出数据时，将此版本号一同读出，之后更新时，对此版本号加1。

          在更新过程中，会对版本号进行比较，如果是一致的，没有发生改变，则会成功执行本次操作；如果版本号不一致，则会更新失败。

2. 对乐观锁的含义有了一定的了解后，结合具体的例子，我们来推演下我们应该怎么处理：

          (1). 假设我们有一张资源表，如下图所示: t\_resource , 其中有6个字段id, resoource,  state, add\_time, update\_time, version,分别表示表主键、资源、分配状态(1未分配  2已分配)、资源创建时间、资源更新时间、资源数据版本号。



         (2). 假设我们现在我们对id=5780这条数据进行分配，那么非分布式场景的情况下，我们一般先查询出来state=1(未分配)的数据，然后从其中选取一条数据可以通过以下语句进行，如果可以更新成功，那么就说明已经占用了这个资源

               update t\_resource set state=2 where state=1 and id=5780。

         (3). 如果在分布式场景中，由于数据库的update操作是原子是原子的，其实上边这条语句理论上也没有问题，但是这条语句如果在典型的“ABA”情况下，我们是无法感知的。有人可能会问什么是“ABA”问题呢？大家可以网上搜索一下，这里我说简单一点就是，如果在你第一次select和第二次update过程中，由于两次操作是非原子的，所以这过程中，如果有一个线程，先是占用了资源(state=2)，然后又释放了资源(state=1)，实际上最后你执行update操作的时候，是无法知道这个资源发生过变化的。也许你会说这个在你说的场景中应该也还好吧，但是在实际的使用过程中，比如银行账户存款或者扣款的过程中，这种情况是比较恐怖的。

         (4). **那么如果使用乐观锁我们如何解决上边的问题呢？**

               a. 先执行select操作查询当前数据的数据版本号,比如当前数据版本号是26：

                   select id, resource, state,version from t\_resource  where state=1 and id=5780;

               b. 执行更新操作：

                   update t\_resoure set state=2, version=27, update\_time=now() where resource=xxxxxx and state=1 and version=26

               c. 如果上述update语句真正更新影响到了一行数据，那就说明占位成功。如果没有更新影响到一行数据，则说明这个资源已经被别人占位了。

      3. 通过2中的讲解，相信大家已经对如何基于数据库表做乐观锁有有了一定的了解了，但是这里还是需要说明一下**基于数据库表做乐观锁的一些缺点:**

          (1). 这种操作方式，使原本一次的update操作，必须变为2次操作: select版本号一次；update一次。增加了数据库操作的次数。

          (2). 如果业务场景中的一次业务流程中，多个资源都需要用保证数据一致性，那么如果全部使用基于数据库资源表的乐观锁，就要让每个资源都有一张资源表，这个在实际使用场景中肯定是无法满足的。而且这些都基于数据库操作，在高并发的要求下，对数据库连接的开销一定是无法忍受的。

          (3). 乐观锁机制往往基于系统中的数据存储逻辑，因此可能会造成脏数据被更新到数据库中。在系统设计阶段，我们应该充分考虑到这些情况出现的可能性，并进行相应调整，如将乐观锁策略在数据库存储过程中实现，对外只开放基于此存储过程的数据更新途径，而不是将数据库表直接对外公开。

使用redis的setnx()、expire()方法，用于分布式锁:

对于使用redis的setnx()、expire()来实现分布式锁，这个方案相对于memcached()的add()方案，redis占优势的是，其支持的数据类型更多，而memcached只支持String一种数据类型。除此之外，无论是从性能上来说，还是操作方便性来说，其实都没有太多的差异，完全看你的选择，比如公司中用哪个比较多，你就可以用哪个。

**首先说明一下setnx()命令，**setnx的含义就是SET if Not Exists，其主要有两个参数 setnx(key, value)。该方法是原子的，如果key不存在，则设置当前key成功，返回1；如果当前key已经存在，则设置当前key失败，返回0。但是要注意的是setnx命令不能设置key的超时时间，只能通过expire()来对key设置。

**具体的使用步骤如下:**

      1. setnx(lockkey, 1)  如果返回0，则说明占位失败；如果返回1，则说明占位成功

      2. expire()命令对lockkey设置超时时间，为的是避免死锁问题。

      3. 执行完业务代码后，可以通过delete命令删除key。

      这个方案其实是可以解决日常工作中的需求的，但从技术方案的探讨上来说，可能还有一些可以完善的地方。**比如，如果在第一步setnx执行成功后，在expire()命令执行成功前，发生了宕机的现象，那么就依然会出现死锁的问题，所以如果要对其进行完善的话，可以使用redis的setnx()、get()和getset()方法来实现分布式锁。**

**使用redis的setnx()、get()、getset()方法，用于分布式锁:**

这个方案的背景主要是在setnx()和expire()的方案上针对可能存在的死锁问题，做了一版优化。

**那么先说明一下这三个命令，对于setnx()和get()这两个命令，相信不用再多说什么。那么getset()命令？**这个命令主要有两个参数 getset(key，newValue)。该方法是原子的，对key设置newValue这个值，并且返回key原来的旧值。假设key原来是不存在的，那么多次执行这个命令，会出现下边的效果：

      1. getset(key, "value1")  返回nil   此时key的值会被设置为value1

      2. getset(key, "value2")  返回value1   此时key的值会被设置为value2

      3. 依次类推！

**介绍完要使用的命令后，具体的使用步骤如下：**

      1. setnx(lockkey, 当前时间+过期超时时间)，如果返回1，则获取锁成功；如果返回0则没有获取到锁，转向2。

      2. get(lockkey)获取值oldExpireTime ，并将这个value值与当前的系统时间进行比较，如果小于当前系统时间，则认为这个锁已经超时，可以允许别的请求重新获取，转向3。

      3. 计算newExpireTime=当前时间+过期超时时间，然后getset(lockkey, newExpireTime) 会返回当前lockkey的值currentExpireTime。

      4. 判断currentExpireTime与oldExpireTime 是否相等，如果相等，说明当前getset设置成功，获取到了锁。如果不相等，说明这个锁又被别的请求获取走了，那么当前请求可以直接返回失败，或者继续重试。

      5. 在获取到锁之后，当前线程可以开始自己的业务处理，当处理完毕后，比较自己的处理时间和对于锁设置的超时时间，如果小于锁设置的超时时间，则直接执行delete释放锁；如果大于锁设置的超时时间，则不需要再锁进行处理。

本文网址：http://www.cnblogs.com/PurpleDream/p/5559352.html