高并发核心技术 - 幂等性 与 分布式锁

2017-08-10

1. 什么是幂等性
   幂等性就是指：一个幂等操作任其执行多次所产生的影响均与一次执行的影响相同。
   用数学的概念表达是这样的： f(f(x)) = f(x).
   就像 nx1 = n 一样， x1 就是一个幂等操作。无论是乘以多少次结果都一样。

2. 常见的幂等性问题
   幂等性问题经常会是由网络问题引起的，还有重复操作引起的。

   场景一：比如点赞功能，一个用户只能对同一片文章点赞一次，重复点赞提示已经点过赞了。
   示例代码：

```
1.  public void like(Article article, User user) {
2.        //检查是否点过赞
3.        if (checkIsLike(article, user)) {
4.            //点过赞了
5.            throw new ApiException(CodeEnums.SYSTEM_ERR);
6.        } else {
7.            //保存点赞
8.            saveLike(article, user);
9.        }
10.   }
```

   看上去好像没有什么问题，保存点赞之前已经检查过是否点赞了，理论上同一个人不会对同一篇文章重复点赞。但实际不是这样的。因为网络请求不是排队进来的，而是一窝蜂涌进来的。
   某些时候，用户网络不好，可能很短的时间内点击了多次，由于网络传输问题，这些请求可能会同时来到我们的服务器。

第一个请求 checkIsLike（） 返回 false ， 正在执行 saveLike() 操作，还没来的及提交事务，
第二个请求过来了 ，checkIsLike（） 返回 也是 false , 并去 执行了 saveLike()

操作。
这样子，就造成了一个用户同时对一篇文章进行了多次点赞操作。

这就是典型的幂等性问题， 操作了一次和操作了两次结果不一样，因为你多点了一次赞，按照幂等性原则 不管你点击了多少次结果都一样，只点了一次赞。

很多场景都是这样造成的，比如用户重复下单，重复评论，重复提交表单等。

那怎么解决呢?
假设网络的请求是排队进来的就不会出现这个问题了。

于是我们可以改成这样:

```
1.     public synchronized void like(Article article, User user) {
2.          //检查是否点过赞
3.          if (checkIsLike(article, user)) {
4.              //点过赞了
5.              throw new ApiException(CodeEnums.SYSTEM_ERR);
6.          } else {
7.              //保存点赞
8.              saveLike(article, user);
9.          }
10.     }
```

synchronized 同步锁 这样我们的请求就会乖乖的排队进来了。

PS： 这样做是效率比较低的做法，不建议这么做，只是举例子，
synchronized 也不适合分布式集群场景。

场景二： 第三方回调

我们系统经常需要和第三方系统打交道，比如微信充值，支付宝充值什么的，微信和支付宝常常会以回调你的接口通知你支付结果。为了保证你能收到回调，往往可能会回调多次。

有时候我们也为了保证数据的准确性会有个定时器去查询支付结果未知的流水，并执行响应的处理。

如果定时器的轮训和回调刚好是在同时进行，这可能又出BUG了,又进行了两次重复操作。

那么问题来了:
假设我是一个充值操作， 回调回来的时候 ，会做业务处理，成功了给用户账户加钱。这是后就要保证幂等性了， 假设微信同一笔交易给你回调了两次，如果你给用户充值了两次，这显然不合理(我是老板肯定扣你工资)，所以要保证 不管微信回调你多少次 ，同一笔交易你只能给用户充一次钱。这就幂等性。

解决幂等性问题方案

- synchronized
  适合单机应用，不追求性能 ，不追求并发。
- 分布式锁
  但是往往我们的应用是分布式的集群，并且很讲究性能，并发，所以我们需要用到 分布式锁 来解决这个问题。

Redis 分布式锁:

```
1.     /**
2.      * setNx
3.      *
4.      * @param key
5.      * @param value
6.      * @return
7.      */
8.     public Boolean setNx(String key, Object value) {
9.         return redisTemplate.opsForValue().setIfAbsent(key, value);
10.    }
11.
12.    /**
13.     * @param key         锁
14.     * @param waitTime    等待时间 毫秒
15.     * @param expireTime 超时时间   毫秒
16.     * @return
17.     */
18.    public Boolean lock(String key, Long waitTime, Long expireTime) {
19.        String vlaue = UUIDUtil.mongoObjectId();
20.        Boolean flag = setNx(key, vlaue);
21.        //尝试获取锁 成功返回
```

```
22.          if (flag) {
23.              redisTemplate.expire(key, expireTime,
    TimeUnit.MILLISECONDS);
24.              return flag;
25.          } else {
26.              //失败
27.              //现在时间
28.              long newTime = System.currentTimeMillis();
29.              //等待过期时间
30.              long loseTime = newTime + waitTime;
31.              //不断尝试获取锁成功返回
32.              while (System.currentTimeMillis() < loseTime) {
33.                  Boolean testFlag = setNx(key, vlaue);
34.                  if (testFlag) {
35.                      redisTemplate.expire(key, expireTime,
    TimeUnit.MILLISECONDS);
36.                      return testFlag;
37.                  }
38.                  //休眠100毫秒
39.                  try {
40.                      Thread.sleep(100);
41.                  } catch (InterruptedException e) {
42.                      e.printStackTrace();
43.                  }
44.              }
45.          }
46.          return false;
47.      }
48.
49.      /**
50.       * @param key
51.       * @return
52.       */
53.      public Boolean lock(String key) {
54.          return lock(key, 1000L, 60 * 1000L);
55.      }
56.
57.      /**
58.       * @param key
59.       */
60.      public void unLock(String key) {
61.          remove(key);
62.      }
```

利用Redis 分布式锁 我们的代码可以改成这样:

```
 1.     public void like(Article article, User user) {
```

```
2.          String key = "key:like" + article.getId() + ":" +
    user.getUserId();
3.          // 等待锁的时间 0   ,  过期时间 一分钟防止死锁
4.          boolean flag = redisService.lock(key, 0, 60 * 1000L);
5.          if(!flag){
6.              //获取锁失败 说明前面的请求已经获取了锁
7.              throw new ApiException(CodeEnums.SYSTEM_ERR);
8.          }
9.          //检查是否点过赞
10.         if (checkIsLike(article, user)) {
11.             //点过赞了
12.             throw new ApiException(CodeEnums.SYSTEM_ERR);
13.         } else {
14.             //保存点赞
15.             saveLike(article, user);
16.         }
17.         //删除锁
18.         redisService.unLock(key);
19.     }
```

key 的设计也很讲究:
数据不冲突的两个业务场景，key不能冲突，不同人的key也不一样，不同的文章Key也不一样。
根据场景业务设定。

一个原则： 尽可能的缩小key的范围。 这样才能增强我们的并发。

首先我们先获取锁，获取锁成功 执行完操作，保存数据 ，删除锁。获取不到锁返回失败。设置过期时间是为了防止‘死锁’，比如机器获取到了 锁，没有设置过期时间，但是他死机了，没有删除释放锁。

- 版本号控制
  CAS 算法： CAS有3个操作数，内存值V，旧的预期值A，要修改的新值B。当且仅当预期值A和内存值V相同时，将内存值V修改为B，否则什么都不做。这个比较繁杂，有兴趣的同学可以去看看。