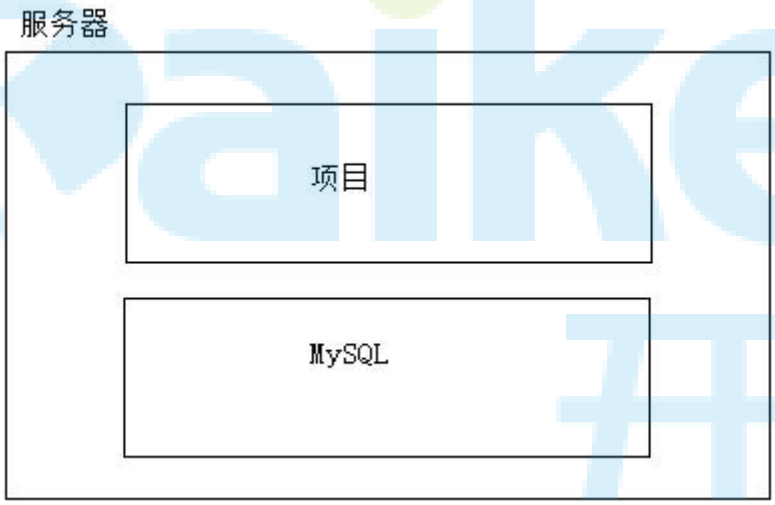
多级缓存&Aop 锁&分布式锁&秒杀下单优化

今日课程主要内容：

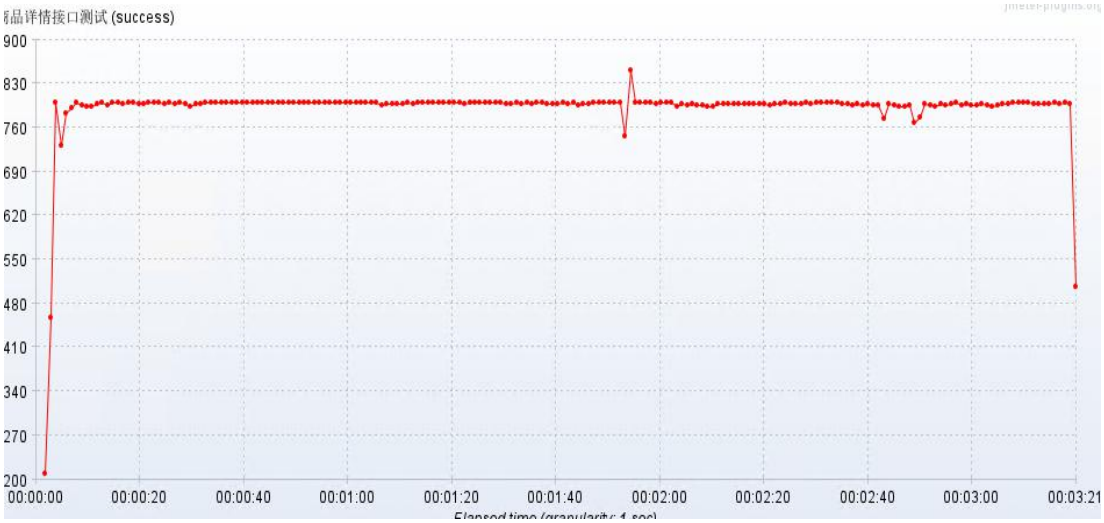
1、分布式部署（TPS 提升能力情况）
2、多级缓存（堆内缓存，分布式缓存，内存字典，lua+redis）
3、秒杀下单业务分析
4、秒杀下单超卖问题—使用锁情况优化

1、分布式部署

1.1 单体结构部署

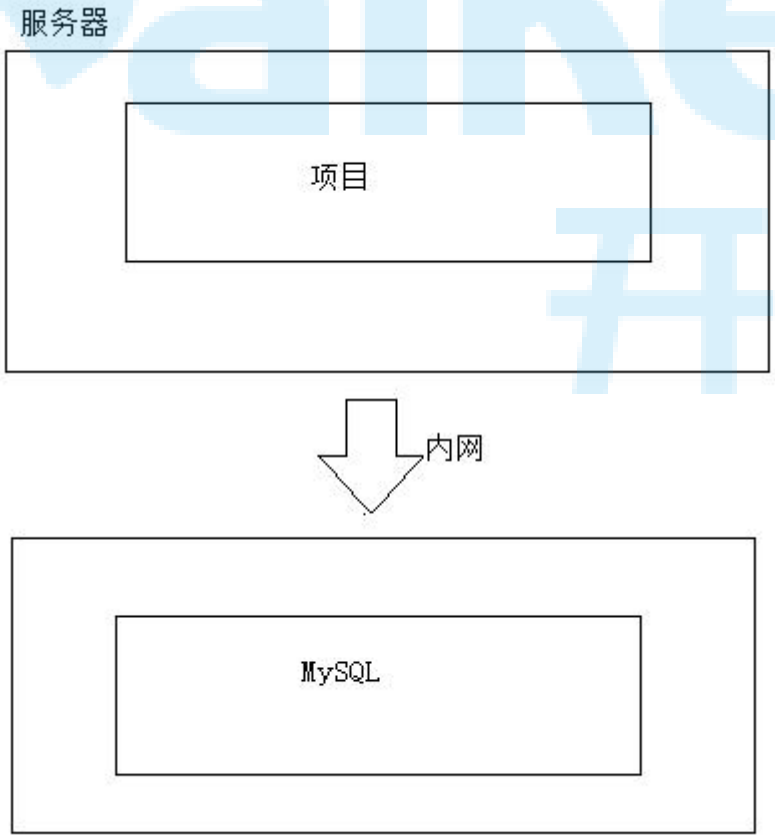


项目服务，MySQL 数据库服务都放在同一个服务器中，在极限压力测试，项目，数据库都会抢占 cpu，内存资源，就意味着项目性能得不到最大的保障；
TPS 性能曲线图如下所示：



1.2 分离模式部署

项目服务，数据库服务进行分离部署；

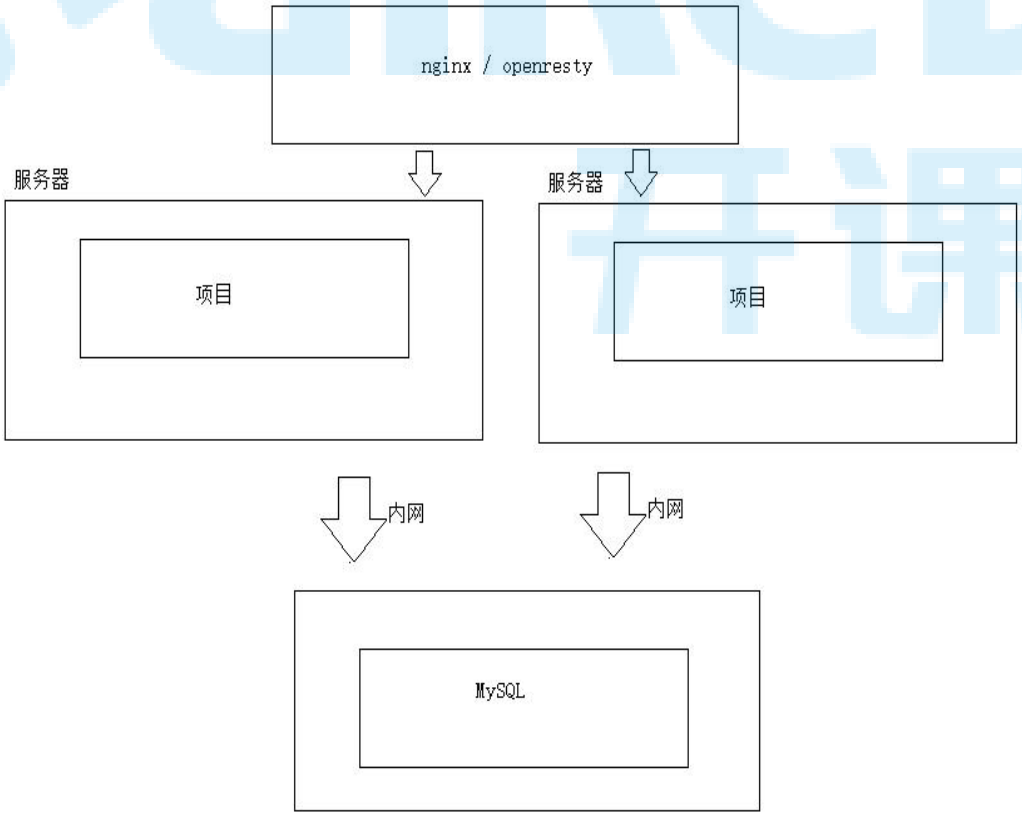


TPS 性能情况如下所示：

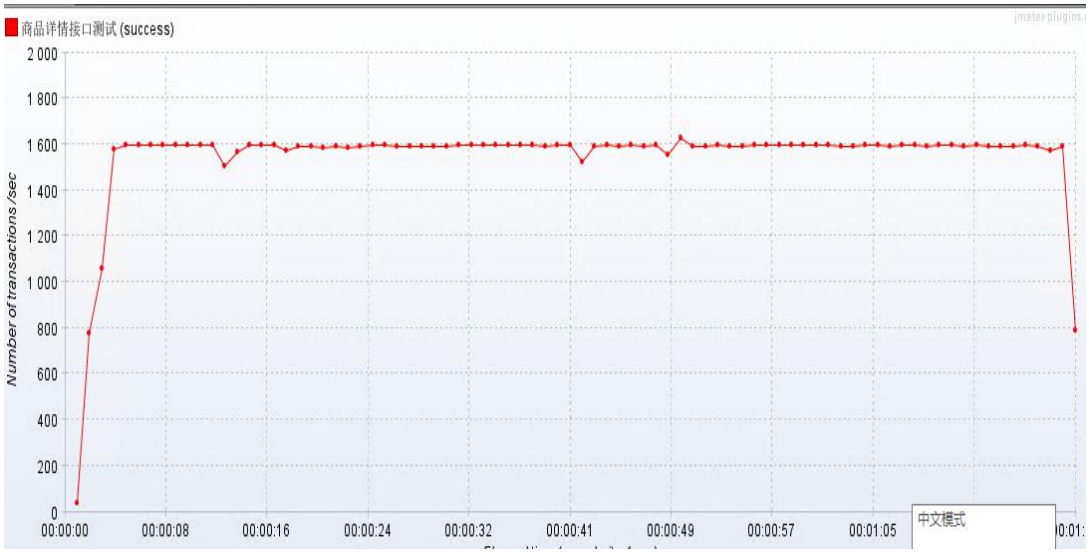
观察测试结果，发现 TPS 性能提升情况效果并不是很明显，因为此操作是一个耗时操作，测试效果不明显；但是 TPS 有一些提升；此处测试 TPS = 800

1.3 集群部署

项目进行集群部署，对项目进行负载均衡的请求的转发；上层使用 openresty 来进行负载均衡；



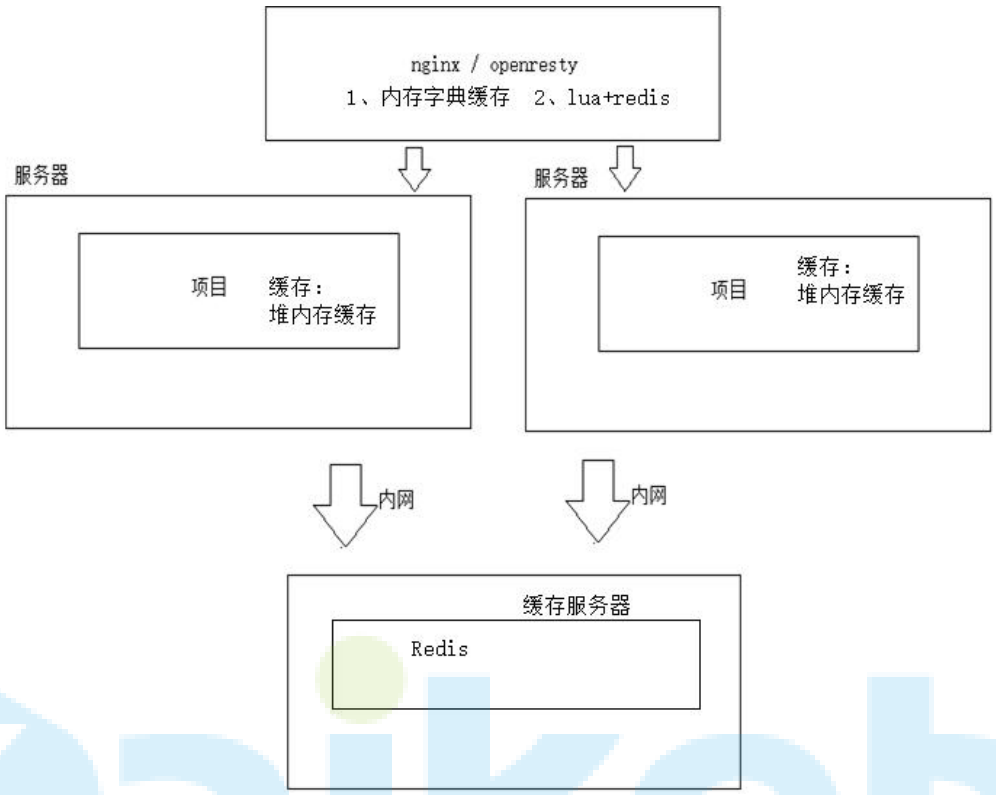
TPS 提升情况：1600 TPS



2 多级缓存

2.1 缓存架构

1、项目进程内部： 堆内存缓存
2、分布式缓存： redis 缓存
3、openresty 接入层缓存：内存字典
4、redis+lua 对缓存结构进行升级



2.2 本地缓存+分布式缓存

缓存问题：

1、本地缓存（堆内存缓存）

采用什么数据结构存储缓存？？

Map 结构：key:value 结构存储缓存数据，数据脏读的问题，堆内存中数据对脏数据是极度不敏感的；

1） JVM 进程内存中，内存资源非常宝贵（java 对象，jvm 对象），JVM 进程级别的缓存，只缓存一些热点的数据

2） JVM 堆内存的资源对脏数据极度不敏感的 （无法实现内存数据和数据库的数据及时的同步）

解决方案： 消息中间件通知进行缓存重写入，定时重写入实现缓存内容更新，但是如此来实现缓存，实现成本稍微有点高，也没有这个必要；

因此解决方案： 给 JVM 内存缓存设置一个超时时间；因此使用 cache 框架；

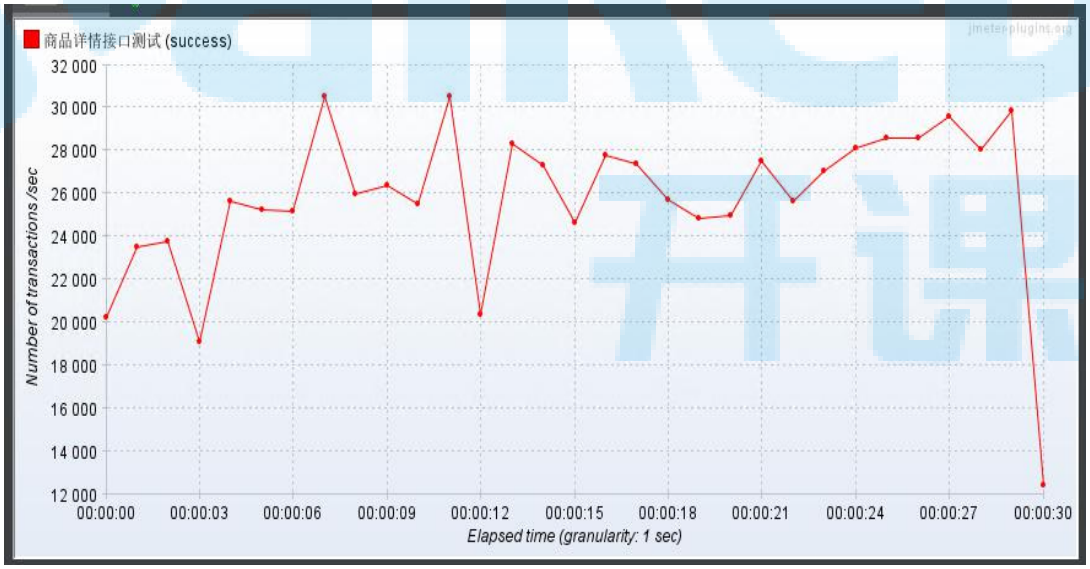
2、分布式缓存

Redis 分布式缓存： AP 模型的 nosql 数据库；

本地缓存+分布式缓存：

```
public TbSeckillGoods findOneByCache(Integer id){

    //1、先从jvm堆缓存中读取数据，使用guva缓存
    TbSeckillGoods seckillGoods = (TbSeckillGoods) guavaCahce.getIfPresent( key: "seckill_goods_"+id);

    //判断jvm堆内缓存是否存在
    if(seckillGoods == null){

        //2、从分布式缓存中查询
        seckillGoods = (TbSeckillGoods) redisTemplate.opsForValue().get("seckill_goods_"+id);

        //判断
        if(seckillGoods == null){
            //3、直接从数据库查询
            seckillGoods = seckillGoodsMapper.selectByPrimaryKey(id);
            if(seckillGoods != null && seckillGoods.getStatus() == 1){
                //添加缓存
                redisTemplate.opsForValue().set( k: "seckill_goods_"+id,seckillGoods, l: 1,TimeUnit.HOURS);
            }
        }

        //添加guava缓存
        guavaCahce.put("seckill_goods_"+id,seckillGoods);
    }

    //如果缓存存在，返回Redis缓存
    return seckillGoods;
```

使用 jvm 堆内存缓存，Redis 分布式缓存后，发现性能得到的极度的飞跃，TPS = 2.8w



2.3 内存字典

内存字典： openresty 内存字典 ---- 实现接入层的缓存 （内存字典： openresty + lua 共同的实现的）；

缓存性能最好： 数据离请求越近的地方，缓存数据的性能越好；

1）openresty 接入 lua 脚本

```
# nginx.conf 配置文件，lua 脚本的接入利用 nginx.conf 文件中 location 接入的；
# content_by_lua ：  直接使用 lua 脚本内容
location /lua1 {
                default_type text/html;
                content_by_lua 'ngx.say("hello lua!!")';
            }

# content_by_lua_file ：  通过文件的方式引入 lua 脚本
location /lua2 {
                default_type text/html;
                content_by_lua_file lua/test.lua;
}
# test.lua
local args = ngx.req.get_uri_args()
ngx.say("hello openresty! lua is so easy!==="..args.id)

# content_by_lua_file  文件模式引入
  location /lua3 {
                content_by_lua_file lua/details.lua;
    }

#details.lua
ngx.exec('/seckill/goods/detail/1');
```

参考资料：https://www.nginx.com/resources/wiki/modules/lua/#directives

- Directives
  - lua_use_default_type
  - lua_code_cache
  - lua_regex_cache_max_entries
  - lua_regex_match_limit
  - lua_package_path
  - lua_package_cpath
  - init_by_lua
  - init_by_lua_file
  - init_worker_by_lua
  - init_worker_by_lua_file
  - set_by_lua
  - set_by_lua_file
  - content_by_lua
  - content_by_lua_file
  - rewrite_by_lua
  - rewrite_by_lua_file
  - access_by_lua
  - access_by_lua_file
  - header_filter_by_lua
  - header_filter_by_lua_file
  - body_filter_by_lua
  - body_filter_by_lua_file

2）实现内存字典缓存

1、开启 openresty 内存字典缓存

```
#proxy_cache_path /usr/local/openresty
lua_shared_dict ngx_cache 128m;
```

2、lua 脚本实现内存字典缓存

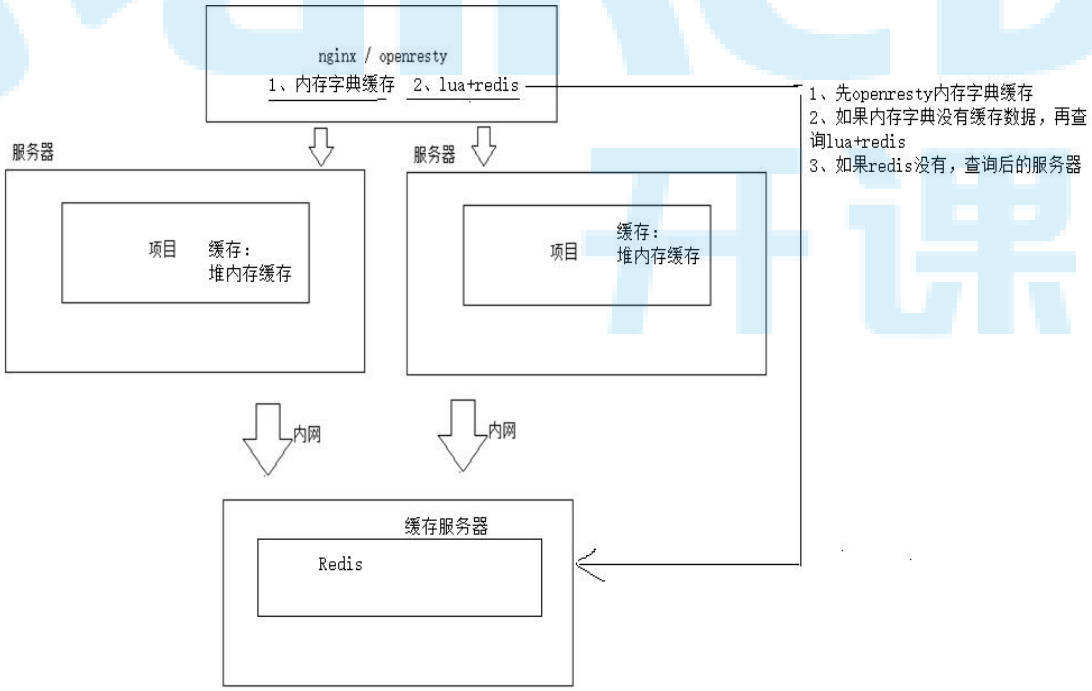
```
-- 基于内存字典实现缓存
-- 添加缓存方法
function set_to_cache(key,value,expritime)
    if not expritime then
        expritime = 0
    end
    -- 获取本地内存字典对象
```

```
    local ngx_cache = ngx.shared.ngx_cache
    -- 添加到本地内存字典数据
    local succ,err,forcible = ngx_cache:set(key,value,expritime)
    return succ;
end

-- 获取内存字典中数据
function get_from_cahce(key)
    -- 获取本地内存字典对象
    local ngx_cache = ngx.shared.ngx_cache
    -- 从内存字典中获取数据
    local vlaue = ngx_cache:get(key)
    return value;
end

-- 简单实现业务逻辑
-- 先从本地内存字典查询数据，如果本地内存字典没有，将会把请求转发给后端服务器，查询数据

-- 先获取请求参数
local params = ngx.req.get_uri_args()
local id = params.id

-- 先从本地内存字典查询数据
local goods = get_from_cahce("seckill_goods_"..id)
-- 判断如果内存字典中没有数据，将会把请求转发给后端服务器
if goods == nil then
    -- 从数据库查询数据
    local res =
ngx.location.capture("/seckill/goods/detail/"..id)
    goods = res.body
    -- 添加到本地内存字典数据
    set_to_cache("seckill_goods_"..id,goods,60)
end
-- 返回结果数据
ngx.say(goods)
```

TPS 性能图形图如下所示：

2.4 Redis+lua

1）缓存架构



Openresty 集成了 redis lua 库，使用 redis+lua 只需要引入 redis lua 库即可使用 redis 相关操作；

```
total 160
-rw-r--r-- 1 root root  6129 Jul 27 14:12 aes.lua
drwxr-xr-x 2 root root  4096 Jul 27 14:12 core
-rw-r--r-- 1 root root   616 Jul 27 14:12 core.lua
drwxr-xr-x 2 root root  4096 Jul 27 14:12 dns
drwxr-xr-x 2 root root  4096 Jul 27 14:12 limit
-rw-r--r-- 1 root root  4682 Jul 27 14:12 lock.lua
drwxr-xr-x 2 root root  4096 Jul 27 14:12 lrucache
-rw-r--r-- 1 root root  7068 Jul 27 14:12 lrucache.lua
-rw-r--r-- 1 root root  1211 Jul 27 14:12 md5.lua
-rw-r--r-- 1 root root 14506 Jul 27 14:12 memcached.lua
-rw-r--r-- 1 root root 21555 Jul 27 14:12 mysql.lua
-rw-r--r-- 1 root root   616 Jul 27 14:12 random.lua
-rw-r--r-- 1 root root 11932 Jul 27 14:12 redis.lua
-rw-r--r-- 1 root root  1192 Jul 27 14:12 sha1.lua
-rw-r--r-- 1 root root  1045 Jul 27 14:12 sha224.lua
-rw-r--r-- 1 root root  1221 Jul 27 14:12 sha256.lua
-rw-r--r-- 1 root root  1045 Jul 27 14:12 sha384.lua
-rw-r--r-- 1 root root  1359 Jul 27 14:12 sha512.lua
-rw-r--r-- 1 root root   236 Jul 27 14:12 sha.lua
-rw-r--r-- 1 root root  4992 Jul 27 14:12 shell.lua
-rwxr-xr-x 1 root root  2854 Jul 27 14:12 signal.lua
-rw-r--r-- 1 root root   731 Jul 27 14:12 string.lua
-rw-r--r-- 1 root root  5178 Jul 27 14:12 upload.lua
drwxr-xr-x 2 root root  4096 Jul 27 14:12 upstream
drwxr-xr-x 2 root root  4096 Jul 27 14:12 websocket
[root@qps001 resty]#
```

引入方式： require “resty.redis”

2）lua 脚本(redis+lua)

```lua
local redis = require "resty.redis"
-- 掉new 方法，获取redis 对象
local red = redis:new()

-- 基于内存字典实现缓存
-- 添加缓存方法
function set_to_cache(key,value,expritime)
    if not expritime then
        expritime = 0
    end
    -- 获取本地内存字典对象
    local ngx_cache = ngx.shared.ngx_cache
    -- 添加到本地内存字典数据
    local succ,err,forcible = ngx_cache:set(key,value,expritime)
    return succ;
end
```

```
-- 获取内存字典中数据
function get_from_cahce(key)
    -- 获取本地内存字典对象
    local ngx_cache = ngx.shared.ngx_cache
    -- 从内存字典中获取数据
    local value = ngx_cache:get(key)

    -- 如果本地内存字典没有
    if not value then
        -- 从 redis 获取缓存数据
        local rev,err = get_to_redis(key)
        if not rev then
            ngx.say("redis cache not exists...",err)
            return
        end
        -- 添加缓存到内存字典
        set_to_cache(key,rev,60)
    end

    return value;
end


-- 向 redis 添加缓存数据
function set_to_redis(key,value)
    -- 设置 redis 超时时间
    red:set_timeout(100000)
    -- 连接 redis 服务器
    local ok,err = red:connect("172.17.61.90",6379)
    -- 判断是否连接成功
    if not ok then
        ngx.say("failed to connect:",err)
        return
    end
    -- 向 redis 添加缓存数据
    local ok,err = red:set(key,value)
    if not ok then
        ngx.say("failed set to redis:",err)
        return
    end
    return ok;
end
```

```
-- 从redis 获取缓存数据
function get_to_redis(key)
    -- 设置redis 超时时间
    red:set_timeout(100000)
    -- 连接redis 服务器
    local ok,err = red:connect("172.17.61.90",6379)
    -- 判断是否连接成功
    if not ok then
        ngx.say("failed to connect:",err)
        return
    end
    -- 从redis 获取缓存数据
    local res,err = red:get(key)
    if not ok then
        ngx.say("failed get to redis:",err)
        return
    end
    ngx.say("get cache from redis.............")
    return res
end



-- 简单实现业务逻辑
-- 先从本地内存字典查询数据，如果本地内存字典没有，将会把请求转发给后端服务器，查询数据

-- 先获取请求参数
local params = ngx.req.get_uri_args()
local id = params.id

-- 先从本地内存字典查询数据
local goods = get_from_cahce("seckill_goods_"..id)
-- 判断如果内存字典中没有数据，将会把请求转发给后端服务器
if goods == nil then
    -- 从数据库查询数据
    local res =
ngx.location.capture("/seckill/goods/detail/"..id)
    goods = res.body
    -- 添加到本地内存字典数据
    set_to_cache("seckill_goods_"..id,goods,60)
end
```

```
-- 返回结果数据
ngx.say(goods)
```

TPS 提升的效果实现：

总结：

主要是针对读操作的优化实现，优化法则：读缓存，写异步

1、服务器优化

2、jvm 优化

3、数据库连接池优化

4、多级缓存

5、服务器分布式部署

伴随着压力测试，观察优化的结果，是否对性能提升有影响；

3 秒杀下单业务分析

3.1 秒杀下单业务实现

秒杀下单业务实现具体逻辑：

1、验证工作（身份信息，token,手机号……，活动是否开始，库存是否充足，是否开启秒杀，是否上架）

2、检查库存（查询库存，是否存在）

3、扣减库存

4、更新库存

5、下单实现

面临问题：

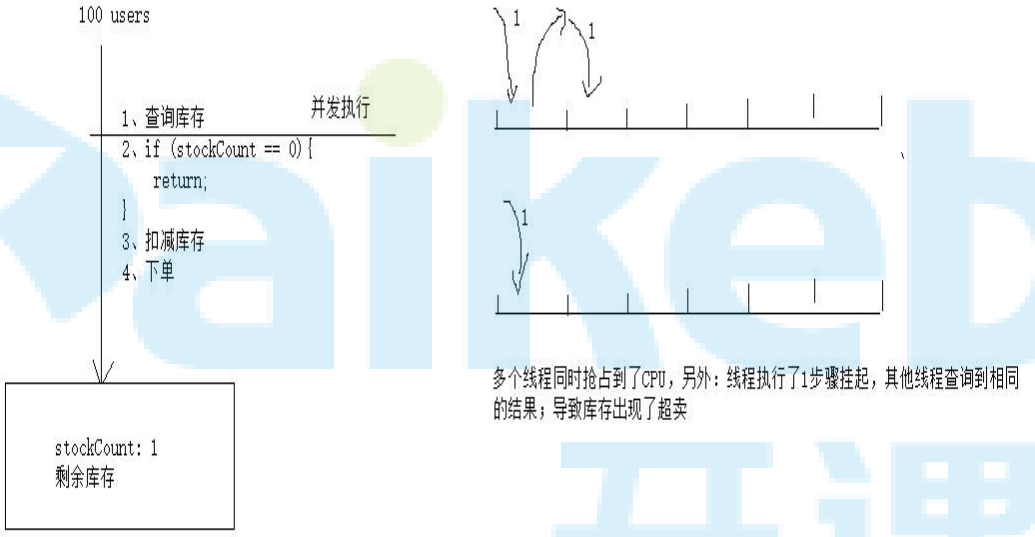
1、业务问题：如何保证库存在高并发模式下，不会出现超卖现象

2、性能问题：如何保证下单操作在高并发模式下，性能问题

3、数据一致性问题： 在高并发模式下，数据一致性问题如何保证

3.2 防止库存超卖

思考题 1： 超卖产生的原因的是什么？？？



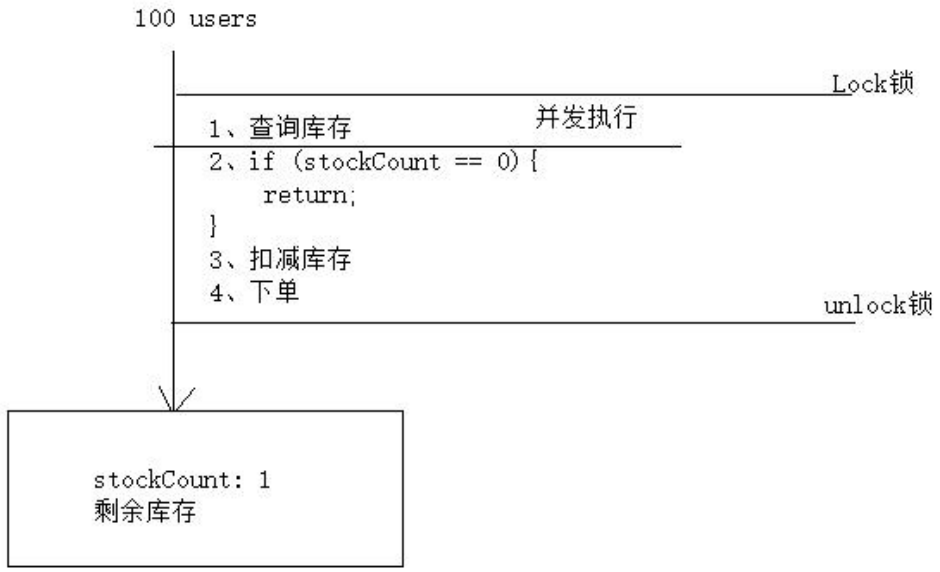
思考题 2： 如何避免出现超卖的问题？？？

解决方案一： 加锁

解决方案二： 原子性操作

解决方案三： 队列

解决方案一： 加锁



使用 lock 锁的方式，只能在单机模式下起作用，在分布式模式下必须使用分布式锁（redis,zookeeper,mysql,etcd）

解决方案二： 原子性操作

Redis 服务器操作具有天然的原子操作的特性，redis 的每一个操作都是一个单线程的操作；因此可以利用 redis 这样的操作模式，来实现库存超卖的问题；

Redis

```
# redis 商品
key: seckill_goods_1
value: {"id":1,name:vivo,stock:2}

# 库存
key: seckill_goods_stock_1
value:2
```

以上存储数据的特点： 把缓存数据单独存储在 redis 服务器中，而不是使用商品的数据字段存储；

此时扣减库存方式：

1、扣减库存： hincrement("seckill_goods_stock_1",-1); ----- 此操作是一个原子操作

2、判断库存是否存在

即兼顾了性能问题，又兼顾业务库存超卖问题；

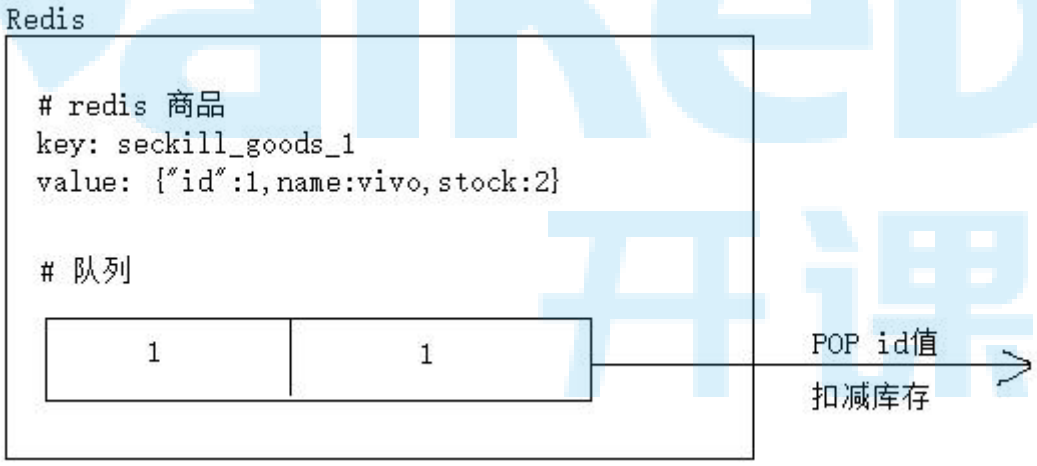
解决方案三： 队列

队列特点：

1、队列的长度等于商品剩余库存数量

2、队列中存储的数据是此商品的 id

3、每一个商品对应一个队列



业务执行：pod 操作也是一个原子性的操作

1、扣减库存： pop 操作从队列中出队一个 ID 值（队列的长度等于库存数量，当队列出队结束，以为库存没了）

2、判断队列长度

4 超卖问题处理

4.1 单机锁

```
@Transactional
@Override
public HttpResult startKilled(Long killId, String userId) {

   try {

      //实现一个加锁的动作
      lock.lock();

      // 从数据库查询商品数据
      TbSeckillGoods seckillGoods =
seckillGoodsMapper.selectByPrimaryKey(killId);
……
      //保存订单
      seckillOrderMapper.insertSelective(order);

      return HttpResult.ok("秒杀成功");
   } catch (Exception e) {
      e.printStackTrace();
   }finally {
      // 防止死锁，必须业务结束后，释放锁
      lock.unlock();
   }
   return null;
}
```

思考： lock 锁，是否可以控制库存超卖现象？？

答案： 不能控制住库存超卖

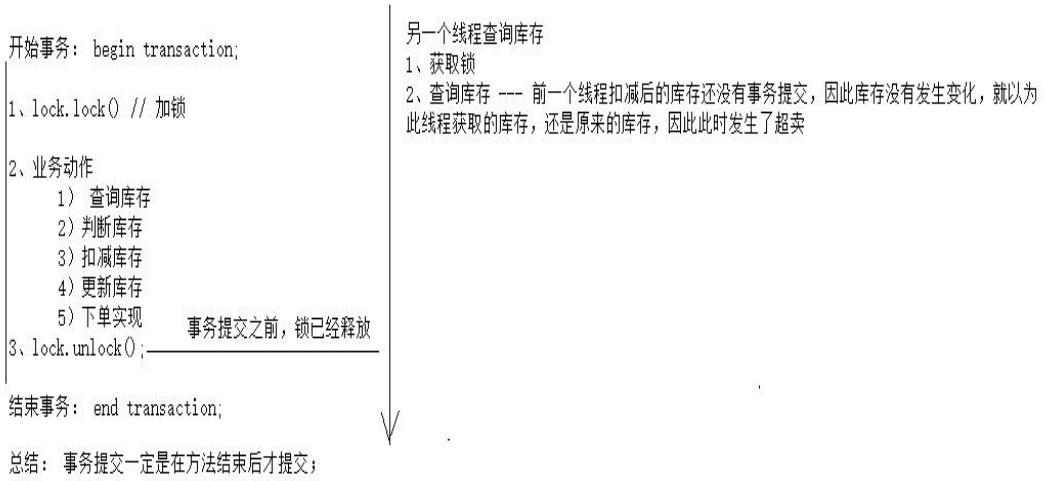
测试实验结果： 1000 库存，1000 线程下单测试

ot_price	give_integral	sort	stock	stock_count	ales	unit_name
(NULL)	(NULL)	1	100	184	66720	节
150.00	0.00	1	100	30	0	节
90.00	0.00	0	100	29	1	节
150.00	0.00	1	100	(NULL)	0	节
90.00	0.00	0	100	(NULL)	0	节

测试结果： 出现了 184 个超卖，说明 lock 锁没有起作用；

原因： 事务和锁冲突

问题 1： 事务在何时提交的？？？

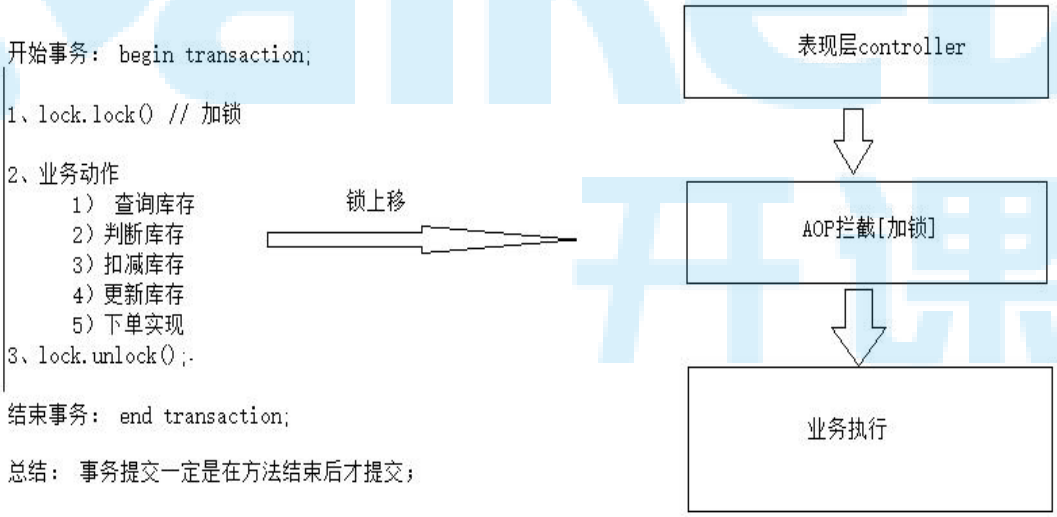


思考题 2 ： 针对以上问题，如何解决这个问题呢？？？

答案：

1、锁上移（controller 表现层）

2、AOP 锁



解决方案： 利用 aop 锁实现锁上移，解决锁和事务冲突的问题；

AOP 锁实现：

```
/**
 * @ClassName ServiceLock
 * @Description 自定义注解，实现 aop 锁
 * @Author hubin
 * @Date 2021/1/23 23:11
 * @Version V1.0
```

```
 **/
@Target({ElementType.PARAMETER,ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface ServiceLock {

    String description() default "";

}
```

```
package com.sugo.seckill.aop;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.context.annotation.Scope;
import org.springframework.core.annotation.Order;
import org.springframework.stereotype.Component;

import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/**
 * @ClassName LockAspect
 * @Description
 * @Author hubin
 * @Date 2021/1/23 23:13
 * @Version V1.0
 **/
@Component
@Scope
@Aspect
@Order(1)
public class LockAspect {

    // 创建 Lock 对象
    private static Lock lock = new ReentrantLock(true);

    // service 切入点
    @Pointcut("@annotation(com.sugo.seckill.aop.ServiceLock)")
    public void lockAspect(){
```

```
    }

    @Around("lockAspect()")
    public Object around(ProceedingJoinPoint joinPoint){

        // 初始化一个对象
        Object obj = null;
        //加锁
        Lock.lock();

        try {
            //执行业务
            obj = joinPoint.proceed();
        } catch (Throwable throwable) {
            throwable.printStackTrace();
        } finally {
            // 业务执行结束后，释放锁
            Lock.unlock();
        }

        return obj;
    }



}
```

经过锁上移后，使用 aop 锁，完美解决了库存超卖的问题；（解决事务和锁冲突问题）

kaikeba
开课吧

kaikeba
开课吧

kaikeba
开课吧