1. redis缓存击穿

批量的缓存信息在同一时间失效，导致请求直接访问数据库信息；

解决方法：保存缓存信息时，在设置**缓存失效时间**时增加随机时间，防止批量缓存在同一时间全部失效。

```
@Transactional
public Product update(Product product) {
    Product productResult = null;
    //RLock updateProductLock = redisson.getLock(LOCK_PRODUCT_UPDATE_PREFIX + product.getId());
    RReadWriteLock readWriteLock = redisson.getReadWriteLock( name: LOCK_PRODUCT_UPDATE_PREFIX + product.getId());
    RLock writeLock = readWriteLock.writeLock();
    writeLock.lock();
    try {
        productResult = productDao.update(product);
        redisUtil.set( key: RedisKeyPrefixConst.PRODUCT_CACHE + productResult.getId(), JSON.toJSONString(productRes
                genProductCacheTimeout(), TimeUnit.SECONDS);
    } finally {
        writeLock.unlock();
    }
    return productResult;
}
```

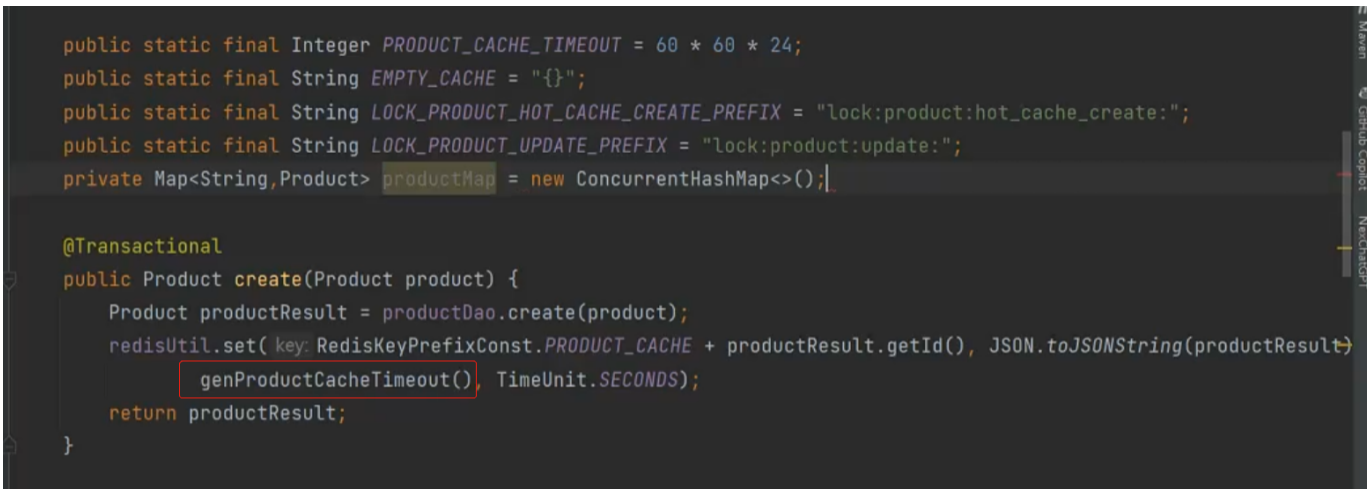
```
public Product get(Long productId) {
    Product product = null;
    String productCacheKey = RedisKeyPrefixConst.PRODUCT_CACHE + productId;

    product = getProductFromCache(productCacheKey);
    if (product != null) {
        return product;
    }
    //DCL
    RLock hotCacheLock = redisson.getLock( name: LOCK_PRODUCT_HOT_CACHE_CREATE_PREFIX + productId);
    hotCacheLock.lock();  //setnx(productId,v)
    try {
        product = getProductFromCache(productCacheKey);
        if (product != null) {
            return product;
        }
```

```
        product = getProductFromCache(productCacheKey);
        if (product != null) {
            return product;
        }

        //RLock updateProductLock = redisson.getLock(LOCK_PRODUCT_UPDATE_PREFIX + productId);
        RReadWriteLock readWriteLock = redisson.getReadWriteLock( name: LOCK_PRODUCT_UPDATE_PREFIX + productId
        RLock rLock = readWriteLock.readLock();
        rLock.lock();
        try {
            product = productDao.get(productId);
            if (product != null) {
                redisUtil.set(productCacheKey, JSON.toJSONString(product),
                        genProductCacheTimeout(), TimeUnit.SECONDS);
            } else {
                redisUtil.set(productCacheKey, EMPTY_CACHE, genEmptyCacheTimeout(), TimeUnit.SECONDS);
            }
        } finally {
            rLock.unlock();
        }
    } finally {
        hotCacheLock.unlock();
    }
    return product;
```

```
private Product getProductFromCache(String productCacheKey) {
    Product product = null;
    String productStr = redisUtil.get(productCacheKey);
    if (!StringUtils.isEmpty(productStr)) {
        if (EMPTY_CACHE.equals(productStr)) {
            return new Product();
        }
        product = JSON.parseObject(productStr, Product.class);
    }
    return product;
}
```



```
public static final Integer PRODUCT_CACHE_TIMEOUT = 60 * 60 * 24;
public static final String EMPTY_CACHE = "{}";
public static final String LOCK_PRODUCT_HOT_CACHE_CREATE_PREFIX = "lock:product:hot_cache_create:";
public static final String LOCK_PRODUCT_UPDATE_PREFIX = "lock:product:update:";
private Map<String,Product> productMap = new ConcurrentHashMap<>();

@Transactional
public Product create(Product product) {
    Product productResult = productDao.create(product);
    redisUtil.set( key: RedisKeyPrefixConst.PRODUCT_CACHE + productResult.getId(), JSON.toJSONString(productResult),
            genProductCacheTimeout(), TimeUnit.SECONDS);
    return productResult;
}
```

2. redis缓存穿透

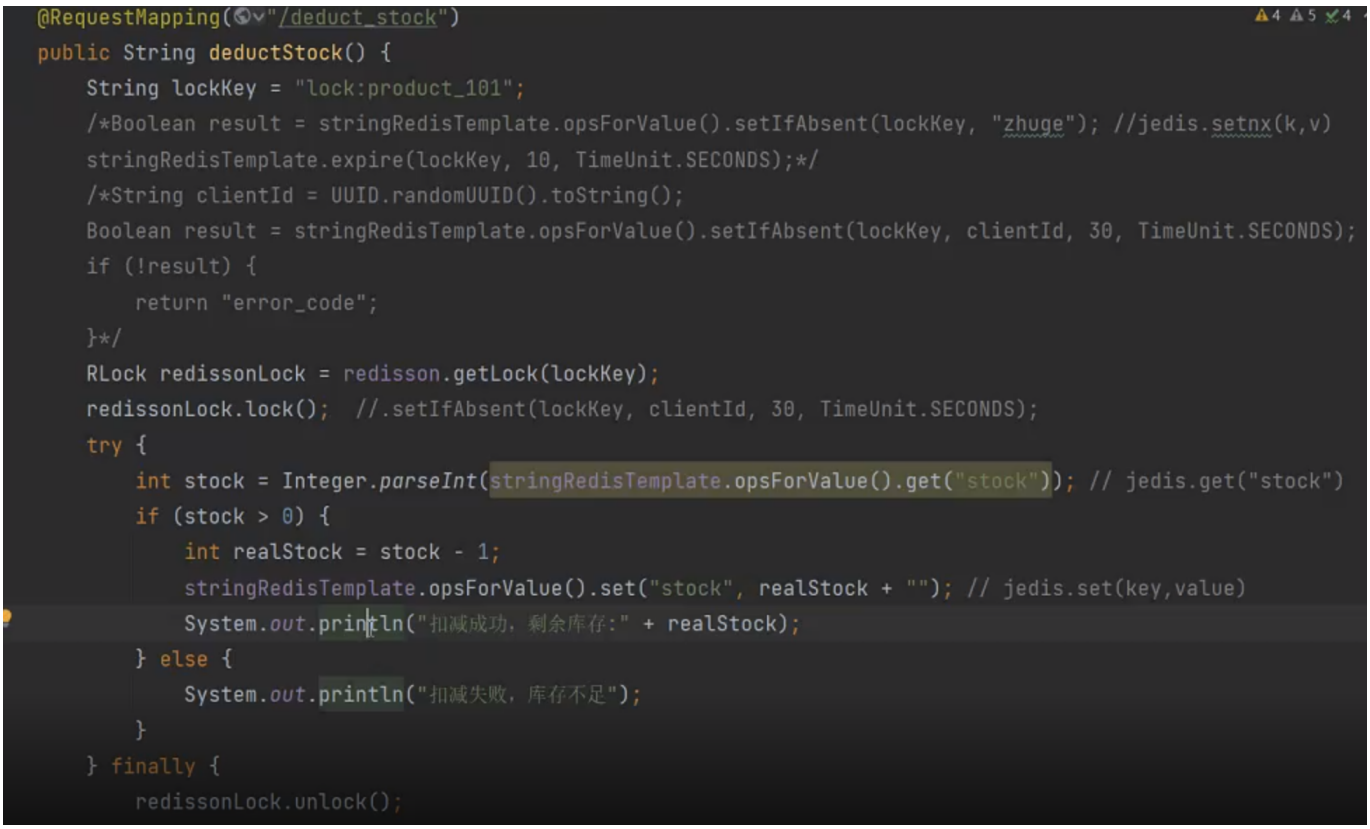
比如redis中某个商品信息被运维人员删除了，然后用户在redis没有查到又去查询数据库，这样把后端的存储层全部查询了一遍。

解决方法：在查询缓存信息时增加空串的判断，在缓存中没有查到返回空串，避免每次在查询缓存中没有查到信息时又去查询数据库；

3. 突发性的热点缓存数据重建

冷门数据没有在redis中保存，突然在秒杀时，大量用户在同一时间排队去访问redis，redis没有查到又排队去查询数据库；

解决方法：加redis分布式锁、双重检测



```java
@RequestMapping("/deduct_stock")
public String deductStock() {
    String lockKey = "lock:product_101";
    /*Boolean result = stringRedisTemplate.opsForValue().setIfAbsent(lockKey, "zhuge"); //jedis.setnx(k,v)
    stringRedisTemplate.expire(lockKey, 10, TimeUnit.SECONDS);*/
    /*String clientId = UUID.randomUUID().toString();
    Boolean result = stringRedisTemplate.opsForValue().setIfAbsent(lockKey, clientId, 30, TimeUnit.SECONDS);
    if (!result) {
        return "error_code";
    }*/
    RLock redissonLock = redisson.getLock(lockKey);
    redissonLock.lock();  //.setIfAbsent(lockKey, clientId, 30, TimeUnit.SECONDS);
    try {
        int stock = Integer.parseInt(stringRedisTemplate.opsForValue().get("stock")); // jedis.get("stock")
        if (stock > 0) {
            int realStock = stock - 1;
            stringRedisTemplate.opsForValue().set("stock", realStock + ""); // jedis.set(key,value)
            System.out.println("扣减成功，剩余库存:" + realStock);
        } else {
            System.out.println("扣减失败，库存不足");
        }
    } finally {
        redissonLock.unlock();
```

缓存雪崩

问题排查

1. 在一个较短的时间内，缓存中较多的key集中过期
2. 此周期内请求访问过期的数据，redis未命中，redis向数据库获取数据
3. 数据库同时接收到大量的请求无法及时处理
4. Redis大量请求被积压，开始出现超时现象
5. 数据库流量激增，数据库崩溃
6. 重启后仍然面对缓存中无数据可用
7. Redis服务器资源被严重占用，Redis服务器崩溃
8. Redis集群呈现崩塌，集群瓦解
9. 应用服务器无法及时得到数据响应请求，来自客户端的请求数量越来越多，应用服务器崩溃
10. 应用服务器，redis，数据库全部重启，效果不理想

问题分析

1. 短时间范围内
2. 大量key集中过期

解决方案（术）

1. 根据业务数据有效期进行分类错峰，A类90分钟，B类80分钟，C类70分钟
2. 过期时间使用固定时间+随机值的形式，稀释集中到期的key的数量

缓存击穿

问题排查

1. Redis中某个key过期，该key访问量巨大
2. 多个数据请求从服务器直接压到Redis后，均未命中
3. Redis在短时间内发起了大量对数据库中同一数据的访问

问题分析

1. 单个key高热数据
2. key过期

解决方案

现场调整

监控访问量，对自然流量激增的数据延长过期时间或设置为永久性key

3. 后台刷新数据

启动定时任务，高峰期来临之前，刷新数据有效期，确保不丢失

4. 二级缓存

设置不同的失效时间，保障不会被同时淘汰就行

5. 加锁

分布式锁，防止被击穿

缓存穿透

问题排查

1. Redis中大面积出现未命中
2. 出现非正常URL访问

问题分析

1. 获取的数据在数据库中也不存在，数据库查询未得到对应数据
2. Redis获取到null数据未进行持久化，直接返回
3. 下次此类数据到达重复上述过程
4. 出现黑客攻击服务器

解决方案（术）

1. 缓存null

对查询结果为null的数据进行缓存（长期使用，定期清理），设定短时限，例如30-60秒，最高5分钟

2. 白名单策略

 提前预热各种分类数据id对应的bitmaps，id作为bitmaps的offset，相当于设置了数据白名单。当加载正常数据时，放

行，加载异常数据时直接拦截（效率偏低）

 使用布隆过滤器（有关布隆过滤器的命中问题对当前状况可以忽略）

3. 实施监控

实时监控redis命中率（业务正常范围时，通常会有一个波动值）与null数据的占比

 非活动时段波动：通常检测3-5倍，超过5倍纳入重点排查对象

 活动时段波动：通常检测10-50倍，超过50倍纳入重点排查对象

根据倍数不同，启动不同的排查流程。然后使用黑名单进行防控（运营）

string 类型数据的基本操作

 添加/修改数据

set key value

 获取数据

get key

 删除数据

del key

append key value

 追加信息到原始信息后部（如果原始信息存在就追加，否则新建）

添加/修改多个数据

mset key1 value1 key2 value2 …

 获取多个数据

mget key1 key2 …

 获取数据字符个数（字符串长度）

strlen key

追加信息到原始信息后部（如果原始信息存在就追加，否则新建）

append key value

incr key

incrby key increment

incrbyfloat key increment

decr key

decrby key increment

hash 类型数据的基本操作

 添加/修改多个数据

hmset key field1 value1 field2 value2 …

 获取多个数据

hmget key field1 field2 ...

 获取哈希表中字段的数量

hlen key

 获取哈希表中是否存在指定的字段

hexists key field

获取哈希表中所有的字段名或字段值

hkeys key

hvals key

 设置指定字段的数值数据增加指定范围的值

hincrby key field increment

hincrbyfloat key field increment

list 类型数据基本操作

lpush key value1 [value2] ……

rpush key value1 [value2] ……

lrange key start stop

lindex key index

llen key

lpop key

rpop key

规定时间内获取并移除数据

blpop key1 [key2] timeout

brpop key1 [key2] timeout

brpoplpush source destination timeout

set 类型数据的基本操作

 添加数据

sadd key member1 [member2]

 获取全部数据

smembers key

 删除数据

srem key member1 [member2]

获取集合数据总量

scard key

 判断集合中是否包含指定数据

sismember key member

随机获取集合中指定数量的数据

srandmember key [count]

随机获取集合中的某个数据并将该数据移出集合

spop key [count]

sorted_set 类型数据的基本操作

添加数据

zadd key score1 member1 [score2 member2]

 获取全部数据

zrange key start stop [WITHSCORES]

zrevrange key start stop [WITHSCORES]

 删除数据

zrem key member [member ...]

按条件获取数据

zrangebyscore key min max [WITHSCORES] [LIMIT]

zrevrangebyscore key max min [WITHSCORES]

按条件删除数据

zremrangebyrank key start stop

zremrangebyscore key min max

 获取集合数据总量

zcard key

zcount key min max

 集合交、并操作

zinterstore destination numkeys key [key ...]

zunionstore destination numkeys key [key ...]

获取数据对应的索引（排名）

zrank key member

zrevrank key member

score值获取与修改

zscore key member

zincrby key increment member