正在加载一言...

一、Redis高并发缓存架构实战

1、Redis集群架构下的分布式锁存在什么问题?

- 主从节点切换可能会丢失redis锁的key

> 比如：有一个主从redis（RedisA作为master，RedisB作为slave）线程一写RedisA成功，返回给客户端（RedisB还没来得及同步key），之后RedisA挂掉了，RedisB切换为主节点；此时线程二请求加锁，可以加锁成功
>
> ZooKeeper可以解决节点切换导致key丢失的问题，但是性能不如redis

- RedLock能解决上述节点切换丢失redis 分布式锁key的问题吗?

> 红锁原理：使用红锁，先要配置多个Redis节点（非集群，彼此之间没有关联，对等的单节点）。加锁的时候对多个Redis节点加锁，多个节点的返回结果要大于半数以上才算加锁成功（比如有3个节点，至少有两个反馈加锁成功，才认为加锁成功，所以第一个问题就来了，多个节点挂掉后，后续加锁操作可能无法进行）。**红锁不推荐使用。**
>
> **红锁的使用:**

```
String lockKey = "product_001";
     //这里需要自己实例化不同redis实例的redisson客户端连接，这里只是伪代码用一个redisson客户端简化了
     RLock lock1 = redisson1.getLock(lockKey);
     RLock lock2 = redisson2.getLock(lockKey);
     RLock lock3 = redisson3.getLock(lockKey);

     /**
          * 根据多个 RLock 对象构建 RedissonRedLock （最核心的差别就在这里）
          */
         RedissonRedLock redLock = new RedissonRedLock(lock1, lock2, lock3);
         try {
             /**
              * waitTimeout 尝试获取锁的最大等待时间，超过这个值，则认为获取锁失败
              * leaseTime   锁的持有时间,超过这个时间锁会自动失效（值应设置为大于业务处理的时间，确保在锁有效期内业务能处理完）
              */
             boolean res = redLock.tryLock(10, 30, TimeUnit.SECONDS);
             if (res) {
                 //成功获得锁，在这里处理业务
             }
         } catch (Exception e) {
             throw new RuntimeException("lock fail");
         } finally {
             //无论如何，最后都要解锁
             redLock.unlock();
         }
         return "end";
```

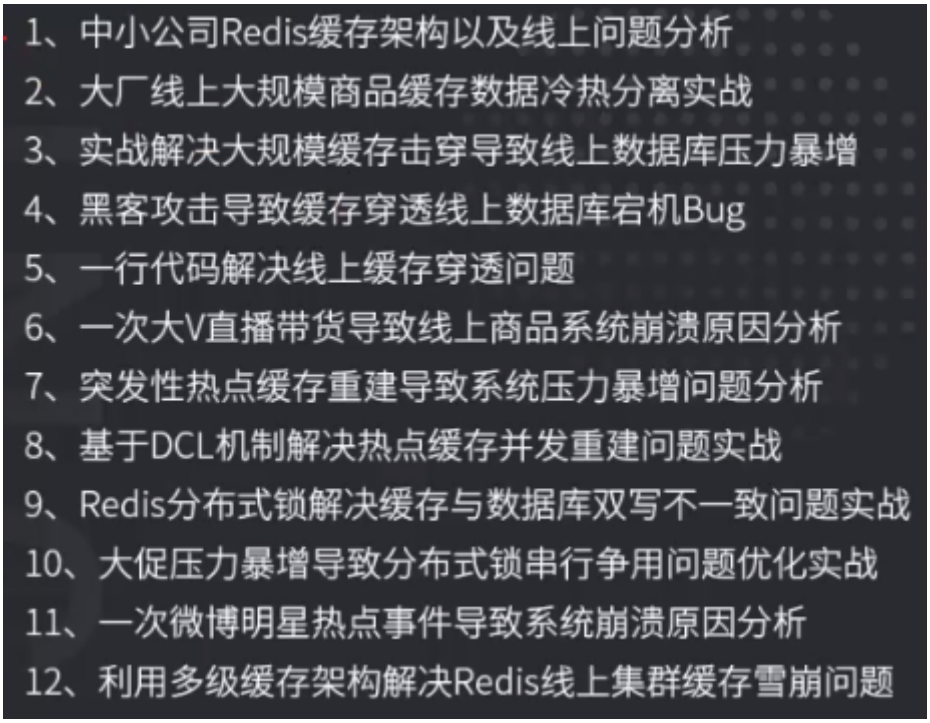
> 红锁存在的问题：
>
> 1> 如果redis每个节点为了保证可用性一般会加从节点；比如有三组Redis（【redis1主、redis2从】、【redis3主、redis4从】、【redis5主、redis6从】）。线程thread1加锁redis1节点成功，加锁thread3节点成功，此时从节点redis4数据还没有同步，然后redis3节点挂了；这个时候有thread2加锁，redis1节点加锁失败（因为被thread1加锁占用）、redis4加锁成功（redis3挂了，redis4就成了主节点，redis4没有同步之前redis3的数据，故可以加锁成功），redis5加锁成功（thread1还没有对thread5加锁）
>
> 可能的解决方法：
>
> ①不加从节点：有问题，当多个主节点挂了，因为加锁成功半数机制存在，可能导致后续所有的锁都无法加锁成功
>
> ②多搞几个节点：性能问题，节点越多，性能越差。用redis本来就是追求高性能，这样适得其反，还不如用zk。
>
> 2> 持久化问题：redis一般会用aof持久化。一般是一秒执行一次持久化。那么究存在问题，这一秒之内的数据可能会丢失，可能导致锁的key丢失，从而造成多个线程对同一个锁key加锁成功，这样就有问题。

2、大促场景下分布式锁性能提升

- 锁的粒度：越小越好，没有必要加锁的代码并行执行
- 分段锁的思想：

> 举例一个秒杀减库存场景：
>
> redis中的key：product_101_stock　value：500（初始库存）
>
> 分段锁的思想是：把这个500个库存分为多个key存储，比如五个
>
> {key:product_101_stock_1　value：100}
>
> {key:product_101_stock_2　value：100}
>
> {key:product_101_stock_3　value：100}
>
> {key:product_101_stock_4　value：100}
>
> {key:product_101_stock_5　value：100}
>
> 由原来的一次性用一个key锁500个库存，改为用5个key分别锁100个库存
>
> 减库存的时候多个线程就可以减不同key的库存，（每个线程来的时候可以随机/轮询选择某个key，当一个key库存减完，就设置一个标记，下次线程就不选这个key了）。1.7的ConcurrentHashMap中就用到了分段锁的思想。

3、大厂的缓存架构实现



- 大规模商品缓存数据冷热分离

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

    String productStr = redisUtil.get(productCacheKey);
    if (!StringUtils.isEmpty(productStr)) {
        product = JSON.parseObject(productStr, Product.class);
        //读延期
        redisUtil.expire(productCacheKey, PRODUCT_CACHE_TIMEOUT, TimeUnit.SECONDS);
        return product;
    }

    product = productDao.get(productId);
    if (product != null) {
        redisUtil.set(productCacheKey, JSON.toJSONString(product),
                PRODUCT_CACHE_TIMEOUT, TimeUnit.SECONDS);
    }
    return product;
}
```

实现了简单的冷热数据分离，常用缓存常驻内

- 缓存击穿（缓存失效）

> 定义：由于大批量缓存在同一时间失效可能导致大量请求同时穿透缓存直达数据库，可能会造成数据库瞬间压力过大甚至挂掉，对于这种情况我们在批量增加缓存时最好**将这一批数据的缓存过期时间设置为一个时间段内的不同时间。**

```
## 过期时间设置为随机数
private Integer genProductCacheTimeout() {
      //加随机超时机制解决缓存批量失效(击穿)问题
      return PRODUCT_CACHE_TIMEOUT + new Random().nextInt(5) * 60 *
60;
}
```

- 缓存穿透

> 缓存穿透是指查询一个根本不存在的数据， 缓存层和存储层都不会命中， 通常出于容错的考虑， 如果从存储层查不到数据则不写入缓存层。

缓存穿透将导致不存在的数据每次请求都要到存储层去查询， 失去了缓存保护后端存储的意义。

造成缓存穿透的基本原因有两个：

第一， 自身业务代码或者数据出现问题。

第二， 一些恶意攻击、 爬虫等造成大量空命中。

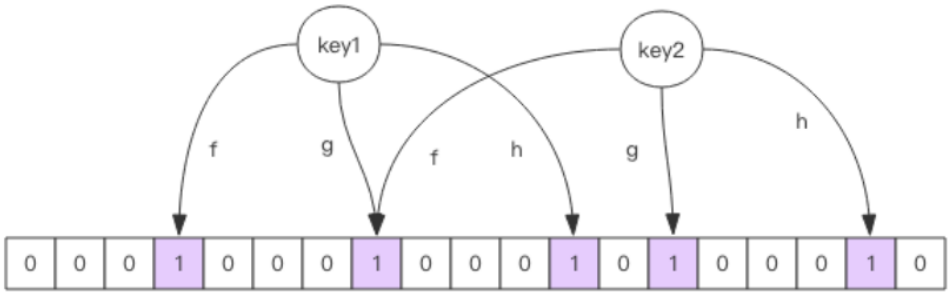
缓存穿透问题解决方案：

- 1、缓存空对象

```
String get(String key) {
    // 从缓存中获取数据
    String cacheValue = cache.get(key);
    // 缓存为空
    if (StringUtils.isBlank(cacheValue)) {
        // 从存储中获取
        String storageValue = storage.get(key);
        cache.set(key, storageValue);
        // 如果存储数据为空， 需要设置一个过期时间(300秒)
        if (storageValue == null) {
            cache.expire(key, 60 * 5);
        }
        return storageValue;
    } else {
        // 缓存非空
        return cacheValue;
    }
}
```

- 2、布隆过滤器

对于恶意攻击，向服务器请求大量不存在的数据造成的缓存穿透，还可以用布隆过滤器先做一次过滤，对于不存在的数据布隆过滤器一般都能够过滤掉，不让请求再往后端发送。当布隆过滤器说**某个值存在时，这个值可能不存在；当它说不存在时，那就肯定不存在。**



布隆过滤器就是**一个大型的位数组和几个不一样的无偏 hash 函数。**所谓无偏就是能够把元素的 hash 值算得比较均匀。

向布隆过滤器中添加 key 时，会使用多个 hash 函数对 key 进行 hash 算得一个整数索引值然后对位数组长度进行取模运算得到一个位置，每个 hash 函数都会算得一个不同的位置。再把位数组的这几个位置都置为 1 就完成了 add 操作。

向布隆过滤器询问 key 是否存在时，跟 add 一样，也会把 hash 的几个位置都算出来，看看位数组中这几个位置是否都为 1，只要有一个位为 0，那么说明布隆过滤器中这个key 不存在。如果都是 1，这并不能说明这个 key 就一定存在，只是极有可能存在，因为这些位被置为 1 可能是因为其它的 key 存在所致。如果这个位数组长度比较

大，存在概率就会很大，如果这个位数组长度比较小，存在概率就会降低。

这种方法适用于数据命中不高、 数据相对固定、 实时性低（通常是数据集较大） 的应用场景， 代码维护较为复杂， 但是**缓存空间占用很少**。

可以用redisson实现布隆过滤器，引入依赖：

```
<dependency>
    <groupId>org.redisson</groupId>
    <artifactId>redisson</artifactId>
    <version>3.6.5</version>
</dependency>
```

代码：

```
package com.redisson;

import org.redisson.Redisson;
import org.redisson.api.RBloomFilter;
import org.redisson.api.RedissonClient;
import org.redisson.config.Config;

public class RedissonBloomFilter {

    public static void main(String[] args) {
        Config config = new Config();

 config.useSingleServer().setAddress("redis://localhost:6379")
;
        //构造Redisson
        RedissonClient redisson = Redisson.create(config);

        RBloomFilter<String> bloomFilter =
redisson.getBloomFilter("nameList");
        //初始化布隆过滤器：预计元素为100000000L,误差率为3%,根据这两个
参数会计算出底层的bit数组大小
        bloomFilter.tryInit(100000000L,0.03);
        //将zhuge插入到布隆过滤器中
        bloomFilter.add("zhuge");

        //判断下面号码是否在布隆过滤器中

 System.out.println(bloomFilter.contains("guojia"));//false

 System.out.println(bloomFilter.contains("baiqi"));//false

 System.out.println(bloomFilter.contains("zhuge"));//true
    }
}
```

注意：布隆过滤器不能删除数据，如果要删除得重新初始化数据。

- 缓存雪崩

缓存雪崩指的是缓存层支撑不住或宕掉后， 流量会像奔逃的野牛一样， 打向后端存储层。

由于缓存层承载着大量请求， 有效地保护了存储层， 但是如果缓存层由于某些原因不能提供服务(比如超大并发过来，缓存层支撑不住，或者由于缓存设计不好，类似大量请求访问bigkey，导致缓存能支撑的并发急剧下降)， 于是大量请求都会打到存储层， 存储层的调用量会暴增， 造成存储层也会级联宕机的情况。

预防和解决缓存雪崩问题， 可以从以下三个方面进行着手。

1） 保证缓存层服务高可用性，比如使用Redis Sentinel或Redis Cluster。

2） 依赖隔离组件为后端限流熔断并降级。比如使用Sentinel或Hystrix限流降级组件。

比如服务降级，我们可以针对不同的数据采取不同的处理方式。当业务应用访问的是非核心数据（例如电商商品属性，用户信息等）时，暂时停止从缓存中查询这些数据，而是直接返回预定义的默认降级信息、空值或是错误提示信息；当业务应用访问的是核心数据（例如电商商品库存）时，仍然允许查询缓存，如果缓存缺失，也可以继续通过数据库读取。

3） 提前演练。 在项目上线前， 演练缓存层宕掉后， 应用以及后端的负载情况以及可能出现的问题， 在此基础上做一些预案设定。

4）多级缓存架构

- 加一级JVM缓存【每秒可以抗百万并发】，这种方案要考虑不同服务器间的jvm缓存的一致性，可以通过mq、redis发布订阅等方式解决。
- 还可能出现短时间的数据不一致，使用了多级缓存，一般这种场景可以容忍。
- 大型互联网公司，还会有热点缓存计算系统（一种可以实现的方式：对redis的数据做aop拦截，然后将redis数据发送到计算系统中，计算系统通过大数据的实时计算系统维护热点缓存，这样web应用系统也做了解耦）【负责put热点数据，web应用系统直接get获取】，所有的web应用会跟这个系统建立联系，比如监听（web应用监听到缓存系统的数据之后，就将数据写入到自己本地缓存）。为什么不web应用自己维护本地缓存，自己put呢？因为数据会变化，可能现在是热点数据，过一会就不是热点数据了。

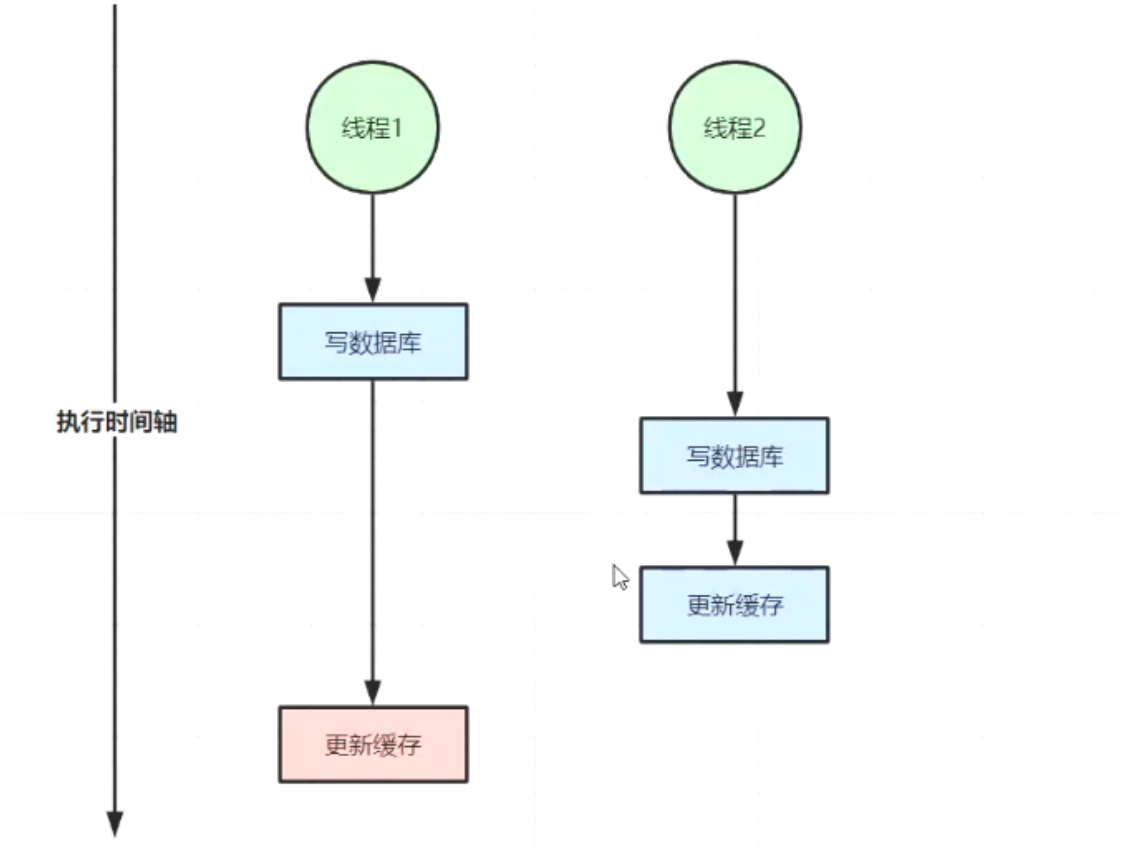
- 突发性热点缓存重建导致系统压力暴增
  - DCL双重检查锁：使用于单机，如果集群环境使用，可能重建缓存会多次
  - 分布式锁重建缓存

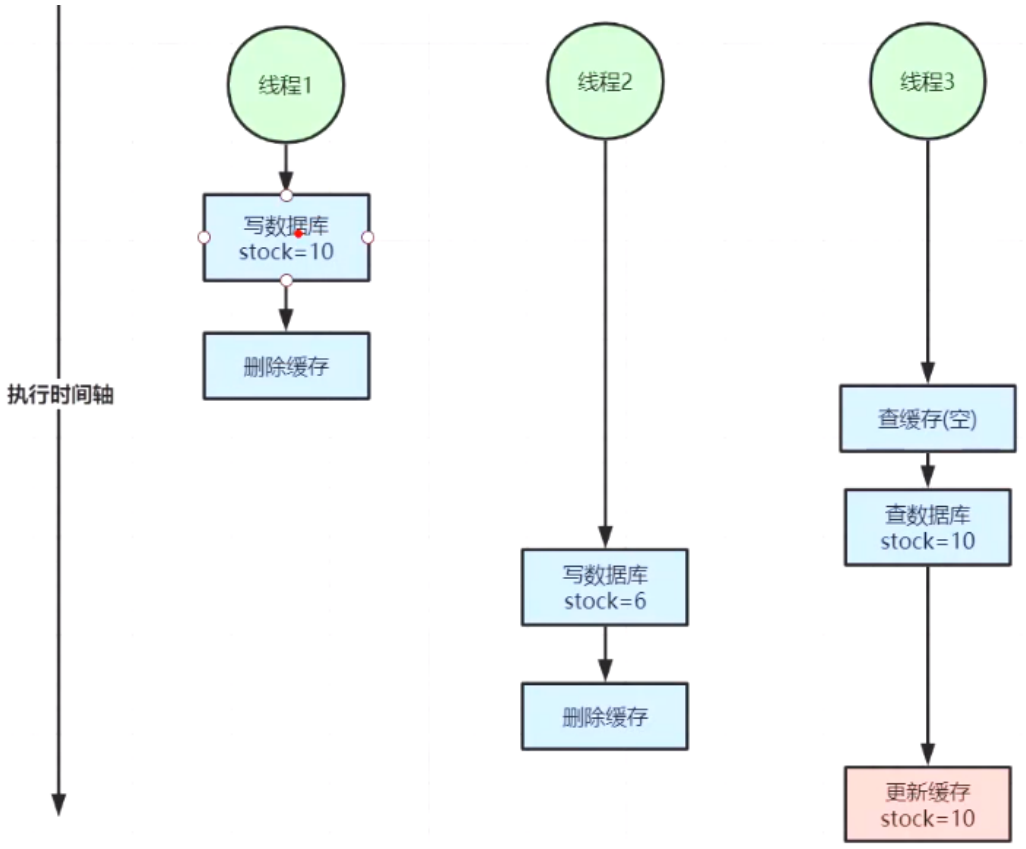
```
public Product get(Long productId) {
        Product product = null;
        String productCacheKey = RedisKeyPrefixConst.PRODUCT_CACHE
+ productId;

        //从缓存里查数据
        product = getProductFromCache(productCacheKey);
        if (product != null) {
            return product;
        }

        //加分布式锁解决热点缓存并发重建问题
        RLock hotCreateCacheLock =
redisson.getLock(LOCK_PRODUCT_HOT_CACHE_CREATE_PREFIX + productId);
        hotCreateCacheLock.lock();
        // 这个优化谨慎使用，防止超时导致的大规模并发重建问题
        // hotCreateCacheLock.tryLock(1, TimeUnit.SECONDS);
        try {
            product = getProductFromCache(productCacheKey);
            if (product != null) {
                return product;
            }

```

```
            //RLock productUpdateLock = redisson.getLock(LOCK_PRODUCT_UPDATE_PREFIX + productId);
            RReadWriteLock productUpdateLock = redisson.getReadWriteLock(LOCK_PRODUCT_UPDATE_PREFIX + productId);
            RLock rLock = productUpdateLock.readLock();
            //加分布式读锁解决缓存双写不一致问题
            rLock.lock();
            try {
                product = productDao.get(productId);
                if (product != null) {
                    redisUtil.set(productCacheKey, JSON.toJSONString(product),
                            genProductCacheTimeout(), TimeUnit.SECONDS);
                } else {
                    //设置空缓存解决缓存穿透问题
                    redisUtil.set(productCacheKey, EMPTY_CACHE, genEmptyCacheTimeout(), TimeUnit.SECONDS);
                }
            } finally {
                rLock.unlock();
            }
        } finally {
            hotCreateCacheLock.unlock();
        }

        return product;
    }
```

- 缓存和数据库双写不一致问题

  原因：





- 加分布式锁

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

        //从缓存里查数据
        product = getProductFromCache(productCacheKey);
        if (product != null) {
            return product;
        }

        //加分布式锁解决热点缓存并发重建问题
        RLock hotCreateCacheLock =
redisson.getLock(LOCK_PRODUCT_HOT_CACHE_CREATE_PREFIX + productId);
        hotCreateCacheLock.lock();
        // 这个优化谨慎使用，防止超时导致的大规模并发重建问题
        // hotCreateCacheLock.tryLock(1, TimeUnit.SECONDS);
        try {
            product = getProductFromCache(productCacheKey);
            if (product != null) {
                return product;
            }

            RLock productUpdateLock =
redisson.getLock(LOCK_PRODUCT_UPDATE_PREFIX + productId);
            // RReadWriteLock productUpdateLock =
redisson.getReadWriteLock(LOCK_PRODUCT_UPDATE_PREFIX + productId);
            // RLock rLock = productUpdateLock.readLock();
            //加分布式读锁解决缓存双写不一致问题
            productUpdateLock.lock();
```

```
            try {
                product = productDao.get(productId);
                if (product != null) {
                    redisUtil.set(productCacheKey, JSON.toJSONString(product),
                                genProductCacheTimeout(), TimeUnit.SECONDS);
                } else {
                    //设置空缓存解决缓存穿透问题
                    redisUtil.set(productCacheKey, EMPTY_CACHE, genEmptyCacheTimeout(), TimeUnit.SECONDS);
                }
            } finally {
                productUpdateLock.unlock();
            }
        } finally {
            hotCreateCacheLock.unlock();
        }

        return product;
    }
```

- 读多写少【主要说的是数据库的读和写】的场景，使用分布式的读写锁，使用Redisson的ReadWriteLock

  如下示例：

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

        //从缓存里查数据
        product = getProductFromCache(productCacheKey);
        if (product != null) {
            return product;
        }

        //加分布式锁解决热点缓存并发重建问题
        RLock hotCreateCacheLock = redisson.getLock(LOCK_PRODUCT_HOT_CACHE_CREATE_PREFIX + productId);
        hotCreateCacheLock.lock();
        // 这个优化谨慎使用，防止超时导致的大规模并发重建问题
        // hotCreateCacheLock.tryLock(1, TimeUnit.SECONDS);
        try {
            product = getProductFromCache(productCacheKey);
            if (product != null) {
                return product;
            }

            //RLock productUpdateLock = redisson.getLock(LOCK_PRODUCT_UPDATE_PREFIX + productId);
            RReadWriteLock productUpdateLock = redisson.getReadWriteLock(LOCK_PRODUCT_UPDATE_PREFIX + productId);
            RLock rLock = productUpdateLock.readLock(); // 加读锁
            //加分布式读锁解决缓存双写不一致问题
            rLock.lock();
```

```
            try {
                product = productDao.get(productId);
                if (product != null) {
                    redisUtil.set(productCacheKey, JSON.toJSONString(product),
                            genProductCacheTimeout(), TimeUnit.SECONDS);
                } else {
                    //设置空缓存解决缓存穿透问题
                    redisUtil.set(productCacheKey, EMPTY_CACHE, genEmptyCacheTimeout(), TimeUnit.SECONDS);
                }
            } finally {
                rLock.unlock();
            }
        } finally {
            hotCreateCacheLock.unlock();
        }

        return product;
    }


## 加写锁
@Transactional
    public Product update(Product product) {
        Product productResult = null;
        //RLock productUpdateLock = redisson.getLock(LOCK_PRODUCT_UPDATE_PREFIX + product.getId());
        RReadWriteLock productUpdateLock = redisson.getReadWriteLock(LOCK_PRODUCT_UPDATE_PREFIX + product.getId());
        RLock writeLock = productUpdateLock.writeLock(); // 加写锁
        //加分布式写锁解决缓存双写不一致问题
        writeLock.lock();
        try {
            productResult = productDao.update(product);
            redisUtil.set(RedisKeyPrefixConst.PRODUCT_CACHE + productResult.getId(), JSON.toJSONString(productResult),
                    genProductCacheTimeout(), TimeUnit.SECONDS);
        } finally {
            writeLock.unlock();
        }
        return productResult;
    }
```

- 缓存重建 加的分布式锁优化====>加个阻塞时间，适用于明确知道执行时间的，否则不要轻易使用

```
##上述讲到了重建缓存加了分布式锁，代码如下
//加分布式锁解决热点缓存并发重建问题
RLock hotCreateCacheLock =
redisson.getLock(LOCK_PRODUCT_HOT_CACHE_CREATE_PREFIX + productId);
hotCreateCacheLock.lock();
// 这个优化谨慎使用，防止超时导致的大规模并发重建问题
// hotCreateCacheLock.tryLock(1, TimeUnit.SECONDS);
try{
    // 业务逻辑
}finally{
    hotCreateCacheLock.unlock();
}
```