第2章 缓存解决方案

学习目标

- 掌握SpringDataRedis 的常用操作
- 能够理解并说出什么是缓存穿透、缓存击穿、缓存雪崩，以及对应的解决方案
- 使用缓存预热的方式实现商品分类导航缓存
- 使用缓存预热的方式实现广告轮播图缓存
- 使用缓存预热的方式实现商品价格缓存

1. SpringDataRedis

1.1 SpringDataRedis简介

SpringDataRedis 属于Spring Data 家族一员，用于对redis的操作进行封装的框架

Spring Data ----- Spring 的一个子项目。Spring 官方提供一套数据层综合解决方案，用于简化数据库访问，支持**NoSQL**和关系数据库存储。包括Spring Data JPA 、Spring Data Redis 、SpringDataSolr 、SpringDataElasticsearch 、Spring DataMongodb 等框架。

1.2 SpringDataRedis快速入门

1.2.1 准备工作

（1）构建Maven工程 SpringDataRedisDemo 引入Spring相关依赖、JUnit依赖、Jedis和SpringDataRedis依赖

```
        <!--缓存-->
        <dependency>
            <groupId>redis.clients</groupId>
            <artifactId>jedis</artifactId>
            <version>2.9.0</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.data</groupId>
            <artifactId>spring-data-redis</artifactId>
            <version>2.0.5.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-test</artifactId>
            <version>5.0.5.RELEASE</version>
        </dependency>
```

（2）在src/main/resources下创建properties文件夹，建立redis-config.properties

```
redis.host=127.0.0.1
redis.port=6379
redis.pass=
redis.database=0
redis.maxIdle=300
redis.maxWait=3000
```

maxIdle ： 最大空闲数

maxWaitMillis: 连接时的最大等待毫秒数

（3）在src/main/resources下创建spring文件夹，创建applicationContext-redis.xml

```
   <context:property-placeholder location="classpath:redis-
config.properties" />
   <!-- redis 相关配置 -->
   <bean id="poolConfig" class="redis.clients.jedis.JedisPoolConfig">
     <property name="maxIdle" value="${redis.maxIdle}" />
     <property name="maxWaitMillis" value="${redis.maxWait}" />
   </bean>
   <bean id="JedisConnectionFactory"
class="org.springframework.data.redis.connection.jedis.JedisConnectionFac
tory"
       p:host-name="${redis.host}" p:port="${redis.port}"
p:password="${redis.pass}" p:pool-config-ref="poolConfig"/>
   <bean id="redisTemplate"
class="org.springframework.data.redis.core.RedisTemplate">
     <property name="connectionFactory" ref="JedisConnectionFactory" />
   </bean>
```

1.2.2 值类型操作

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations="classpath:spring/applicationContext-
redis.xml")
public class TestValue {
    @Autowired
    private RedisTemplate redisTemplate;

    @Test
    public void setValue(){
        redisTemplate.boundValueOps("name").set("itcast");
    }

    @Test
    public void getValue(){
        String str = (String) redisTemplate.boundValueOps("name").get();
        System.out.println(str);
    }

    @Test
    public void deleteValue(){
        redisTemplate.delete("name");;
    }
}
```

1.2.3 Set类型操作

```java
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations="classpath:spring/applicationContext-redis.xml")
public class TestSet {

    @Autowired
    private RedisTemplate redisTemplate;

    /**
     * 存入值
     */
    @Test
    public void setValue(){
        redisTemplate.boundSetOps("nameset").add("曹操");
        redisTemplate.boundSetOps("nameset").add("刘备");
        redisTemplate.boundSetOps("nameset").add("孙权");
    }

    /**
     * 提取值
     */
    @Test
    public void getValue(){
        Set members = redisTemplate.boundSetOps("nameset").members();
        System.out.println(members);
    }

    /**
     * 删除集合中的某一个值
     */
    @Test
    public void deleteValue(){
        redisTemplate.boundSetOps("nameset").remove("孙权");
    }

    /**
     * 删除整个集合
     */
    @Test

    public void deleteAllValue(){
```

```
        redisTemplate.delete("nameset");
    }
}
```

1.2.4 List类型操作

（1）右压栈 后添加的对象排在后边

```
    /**
     * 右压栈：后添加的对象排在后边
     */
    @Test
    public void testSetValue1(){
        redisTemplate.boundListOps("namelist1").rightPush("刘备");
        redisTemplate.boundListOps("namelist1").rightPush("关羽");
        redisTemplate.boundListOps("namelist1").rightPush("张飞");

    }

    /**
     * 显示右压栈集合
     */
    @Test
    public void testGetValue1(){
        List list = redisTemplate.boundListOps("namelist1").range(0, 10);
        System.out.println(list);
    }
```

运行结果：

[刘备, 关羽, 张飞]

（2）左压栈 后添加的对象排在前边

```
    /**
     * 左压栈：后添加的对象排在前边
     */
    @Test
    public void testSetValue2(){
        redisTemplate.boundListOps("namelist2").leftPush("刘备");
        redisTemplate.boundListOps("namelist2").leftPush("关羽");
        redisTemplate.boundListOps("namelist2").leftPush("张飞");
    }

    /**
     * 显示左压栈集合
     */
    @Test
    public void testGetValue2(){
        List list = redisTemplate.boundListOps("namelist2").range(0, 10);
        System.out.println(list);
    }
```

运行结果：

[张飞, 关羽, 刘备]

（3）根据索引查询元素

```
    /**
     * 查询集合某个元素
     */
    @Test
    public void testSearchByIndex(){
        String s = (String)
redisTemplate.boundListOps("namelist1").index(1);
        System.out.println(s);
    }
```

（4）移除指定个数的值

```
    /**
     * 移除集合某个元素
     */
    @Test
    public void testRemoveByIndex(){
        redisTemplate.boundListOps("namelist1").remove(1, "关羽");
    }
```

1.2.5 Hash类型操作

（1）存入值

```
    @Test
    public void testSetValue(){
        redisTemplate.boundHashOps("namehash").put("a", "唐僧");
        redisTemplate.boundHashOps("namehash").put("b", "悟空");
        redisTemplate.boundHashOps("namehash").put("c", "八戒");
        redisTemplate.boundHashOps("namehash").put("d", "沙僧");
    }
```

（2）提取所有的KEY

```
    @Test
    public void testGetKeys(){
        Set s = redisTemplate.boundHashOps("namehash").keys();
        System.out.println(s);
    }
```

运行结果：

[a, b, c, d]

（3）提取所有的值

```
    @Test
    public void testGetValues(){
        List values = redisTemplate.boundHashOps("namehash").values();
        System.out.println(values);
    }
```

运行结果：

[唐僧, 悟空, 八戒, 沙僧]

（4）根据KEY提取值

```
    @Test
    public void testGetValueByKey(){
        Object object = redisTemplate.boundHashOps("namehash").get("b");
        System.out.println(object);
    }
```

运行结果：

悟空

（5）根据KEY移除值

```
    @Test
    public void testDeleteValueByKey(){
        redisTemplate.boundHashOps("namehash").delete("c");
    }
```

运行后再次查看集合内容：

[唐僧, 悟空, 沙僧]

1.2.6 ZSet类型操作

zset是set的升级版本，它在set的基础上增加了一格顺序属性，这一属性在添加元素的同时可以指定，每次指定后，zset会自动重新按照新的值调整顺序。可以理解为有两列的mysql表，一列存储value，一列存储分值。

（1）存值，指定分值

```
    @Test
    public void setValue(){
        redisTemplate.boundZSetOps("namezset").add("曹操",100000);
        redisTemplate.boundZSetOps("namezset").add("孙权",0);
        redisTemplate.boundZSetOps("namezset").add("刘备",1000);
    }
```

（2）查询，由低到高

```
    /**
     * 由低到高排序
     */
    @Test
    public void getValue(){
        Set namezset = redisTemplate.boundZSetOps("namezset").range(0,
-1);
        System.out.println(namezset);
    }
```

（3）查询，由高到低，土豪榜前10

```
    /**
     * 由高到底排序（土豪榜）
     */
    @Test
    public void tuhaobang(){
        Set namezset =
redisTemplate.boundZSetOps("namezset").reverseRange(0,9);
        System.out.println(namezset);
    }
```

（4）增加分数

```
    /**
     * 增加分值
     */
    @Test
    public void addScore(){
        redisTemplate.boundZSetOps("namezset").incrementScore("孙
权",2000);
    }
```

（5）查询值和分数

```
    /**
     * 查询值和分数
     */
    @Test
    public void getValueAndScore(){
        Set<ZSetOperations.TypedTuple> namezset =
redisTemplate.boundZSetOps("namezset").reverseRangeWithScores(0, -1);
        for(ZSetOperations.TypedTuple typedTuple:namezset){
            System.out.print("姓名: "+typedTuple.getValue());
            System.out.println("金币: "+typedTuple.getScore());
        }
    }
```

TypedTuple是值与分数的封装。

1.2.7 过期时间设置

以值类型为例：存值时指定过期时间和时间单位

```
    /**
     * 存值
     */
    @Test
    public void setValue(){
        redisTemplate.boundValueOps("name").set("itcast");
        redisTemplate.boundValueOps("name").expire(10,TimeUnit.SECONDS);
    }
```

2. 缓存穿透、缓存击穿、缓存雪崩

2.1 缓存穿透

缓存穿透是指缓存和数据库中都没有的数据，而用户不断发起请求，如发起为id为“-1”的数据或id为特别大不存在的数据。这时的用户很可能是攻击者，攻击会导致数据库压力过大。如下面这段代码就存在缓存穿透的问题。

```
    public Integer findPrice(Long id) {
        //从缓存中查询
        Integer sku_price =
(Integer)redisTemplate.boundHashOps("sku_price").get(id);
        if(sku_price==null){
            //缓存中没有，从数据库查询
            Sku sku = skuMapper.selectByPrimaryKey(id);
            if(sku!=null){ //如果数据库有此对象
                sku_price = sku.getPrice();

redisTemplate.boundHashOps("sku_price").put(id,sku_price);
            }
        }
        return sku_price;
    }
```

解决方案：

1.接口层增加校验，如用户鉴权校验，id做基础校验，id<=0的直接拦截；

2.从缓存取不到的数据，在数据库中也没有取到，这时也可以将key-value对写为key-0。这样可以防止攻击用户反复用同一个id暴力攻击。代码举例：

```
    public int findPrice(Long id) {
        //从缓存中查询
        Integer sku_price =
(Integer)redisTemplate.boundHashOps("sku_price").get(id);
        if(sku_price==null){
            //缓存中没有，从数据库查询
            Sku sku = skuMapper.selectByPrimaryKey(id);
            if(sku!=null){ //如果数据库有此对象
                sku_price = sku.getPrice();

redisTemplate.boundHashOps("sku_price").put(id,sku_price);
            }else{
                redisTemplate.boundHashOps("sku_price").put(id,0);
            }
        }
        return sku_price;
    }
```

3. 使用缓存预热

缓存预热就是将数据提前加入到缓存中，当数据发生变更，再将最新的数据更新到缓存。

后边我们就用缓存预热的方式实现对分类导航、广告轮播图等数据的缓存。

2.2 缓存击穿

缓存击穿是指缓存中没有但数据库中有的数据。这时由于并发用户特别多，同时读缓存没读到数据，又同时去数据库去取数据，引起数据库压力瞬间增大，造成过大压力。

以下代码可能会产生缓存击穿：

```
    @Autowired
    private RedisTemplate redisTemplate;

    public List<Map> findCategoryTree() {
        //从缓存中查询
        List<Map> categoryTree=
(List<Map>)redisTemplate.boundValueOps("categoryTree").get();
        if(categoryTree==null){
            Example example=new Example(Category.class);
            Example.Criteria criteria = example.createCriteria();
            criteria.andEqualTo("isShow","1");//显示
            List<Category> categories =
categoryMapper.selectByExample(example);
            categoryTree=findByParentId(categories,0);

redisTemplate.boundValueOps("categoryTree").set(categoryTree);
            //过期时间设置  ......
        }
        return categoryTree;
    }
```

解决方案：

1.设置热点数据永远不过期。

2.缓存预热

2.3 缓存雪崩

缓存雪崩是指缓存数据大批量到过期时间，而查询数据量巨大，引起数据库压力过大甚至down机。和缓存击穿不同的是，缓存击穿指并发查同一条数据，缓存雪崩是不同数据都过期了，很多数据都查不到从而查数据库。

解决方案：

1.缓存数据的过期时间设置随机，防止同一时间大量数据过期现象发生。

2.设置热点数据永远不过期。

3.使用缓存预热

3. 商品分类导航缓存

3.1 需求分析

为了提升首页的加载速度，减轻数据库访问压力，我们将首页的商品分类导航数据加载在缓存中。

3.2 实现思路

为了避免缓存穿透、击穿等问题，我们采用缓存预热的方式实现对分类导航数据的缓存。

考虑到商品分类导航数据不经常变动，所以我们不设置过期时间。

3.3 代码实现

3.3.1 通用模块整合spring data redis

（1）qingcheng_common_service引入依赖

```
        <!--缓存-->
        <dependency>
            <groupId>redis.clients</groupId>
            <artifactId>jedis</artifactId>
            <version>2.9.0</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.data</groupId>
            <artifactId>spring-data-redis</artifactId>
            <version>2.0.5.RELEASE</version>
        </dependency>
```

（2）qingcheng_common_service新增配置文件 redis-config.properties

```
redis.host=127.0.0.1
redis.port=6379
redis.pass=
redis.database=0
redis.maxIdle=300
redis.maxWait=3000
```

maxWait：连接池中连接用完时,新的请求等待时间,毫秒

maxIdle： 最大闲置个数

（3）qingcheng_common_service新增spring配置文件applicationContext-redis.xml

```
    <!-- redis 相关配置 -->
    <bean id="poolConfig" class="redis.clients.jedis.JedisPoolConfig">
      <property name="maxIdle" value="${redis.maxIdle}" />
      <property name="maxWaitMillis" value="${redis.maxWait}" />
    </bean>
    <bean id="jedisConnectionFactory"
class="org.springframework.data.redis.connection.jedis.JedisConnectionFac
tory"
        p:host-name="${redis.host}" p:port="${redis.port}"
p:password="${redis.pass}" p:pool-config-ref="poolConfig"/>
    <bean id="redisTemplate"
class="org.springframework.data.redis.core.RedisTemplate">
        <property name="connectionFactory" ref="jedisConnectionFactory"
/>
    </bean>
```

（4）qingcheng_common_service工程新增枚举

```
public enum CacheKey {

    AD,//广告
    SKU_PRICE,//价格
    CATEGORY_TREE;//商品分类导航树
}
```

3.3.2 商品分类加载到缓存

（1）服务接口CategoryService新增方法定义

```
    /**
     * 将商品分类树放入缓存
     */
    public void saveCategoryTreeToRedis();
```

（2）CategoryServiceImpl实现此方法

```
    @Autowired
    private RedisTemplate redisTemplate;

    public void saveCategoryTreeToRedis() {
        System.out.println("加载商品分类数据到缓存");
        Example example=new Example(Category.class);
        Example.Criteria criteria = example.createCriteria();
        criteria.andEqualTo("isShow","1");//显示
        List<Category> categories =
categoryMapper.selectByExample(example);
        List<Map> categoryTree =findByParentId(categories,0);

redisTemplate.boundValueOps(CacheKey.CATEGORY_TREE).set(categoryTree);
    }
```

（3）qingcheng_service_goods工程新增类

```
@Component
public class Init implements InitializingBean {

    @Autowired
    private CategoryService categoryService;

    public void afterPropertiesSet() throws Exception {
        System.out.println("---缓存预热---");
        categoryService.saveCategoryTreeToRedis();//加载商品分类导航缓存
    }
}
```

实现InitializingBean接口的类会在启动时自动调用。

3.3.3 查询商品分类缓存

修改CategoryServiceImpl的findCategoryTree方法， 直接从缓存中提取数据。

```
   public List<Map> findCategoryTree() {
        //从缓存中查询
        return
(List<Map>)redisTemplate.boundValueOps(CacheKey.CATEGORY_TREE).get();

   }
```

3.3.4 更新商品分类缓存

修改CategoryServiceImpl的增删改方法，在增删改后重新加载缓存

```
    /**
     * 新增
     * @param category
     */
    public void add(Category category) {
        categoryMapper.insert(category);
        saveCategoryTreeToRedis();
    }

    /**
     * 修改
     * @param category
     */
    public void update(Category category) {
        categoryMapper.updateByPrimaryKeySelective(category);
        saveCategoryTreeToRedis();
    }

    /**
     *  删除
     * @param id
     */
    public void delete(Integer id) {
        //判断是否存在下级分类
        //......
        saveCategoryTreeToRedis();
    }
```

4. 广告轮播图缓存

4.1 需求分析

为了提升首页的加载速度，减轻数据库访问压力，我们将首页的广告轮播图数据加载在缓存中。

4.2 实现思路

使用“缓存预热”的方式实现

广告数据不只是轮播图，我们可以使用hash来存储广告数据。

4.3 代码实现

4.3.1 广告数据加载到缓存

（1）AdService新增方法定义

```
    /**
     * 将某个位置的广告存入缓存
     * @param position
     */
    public void saveAdToRedisByPosition(String position);


    /**
     * 将全部广告数据存入缓存
     */
    public void saveAllAdToRedis();
```

（2）AdServiceImpl方法实现

```
    @Autowired
    private RedisTemplate redisTemplate;

    public void saveAdToRedisByPosition(String position) {
        //查询某位置的广告列表
        Example example=new Example(Ad.class);
        Example.Criteria criteria = example.createCriteria();
        criteria.andEqualTo("position",position);
        criteria.andLessThanOrEqualTo("startTime",new Date());//开始时间小
于等于当前时间
        criteria.andGreaterThanOrEqualTo("endTime",new Date());//截至时间
大于等于当前时间
        criteria.andEqualTo("status","1");
        List<Ad> adList = adMapper.selectByExample(example);
        //装入缓存
        redisTemplate.boundHashOps(CacheKey.AD).put(position,adList);
    }

    /**
     * 返回所有的广告位置
     * @return
     */
    private List<String> getPositionList(){
        List<String> positionList=new ArrayList<String>();
        positionList.add("index_lb");//首页广告轮播图
        //。。。

        return positionList;
    }

    public void saveAllAdToRedis() {
        //循环所有的广告位置
        for(String position:getPositionList()){
            saveAdToRedisByPosition(position);
        }
    }
```

（3）qingcheng_service_business工程新增类

```
@Component
public class Init implements InitializingBean {

    @Autowired
    private AdService adService;

    public void afterPropertiesSet() throws Exception {
        System.out.println("---缓存预热---");
        adService.saveAllAdToRedis();
    }
}
```

4.3.2 查询广告缓存

修改AdServiceImpl的findByPosition方法

```
    public List<Ad> findByPosition(String position) {
        //从缓存中查询广告列表

return(List<Ad>)redisTemplate.boundHashOps(CacheKey.AD).get(position);
    }
```

4.3.3 更新广告缓存

修改AdServiceImpl的增删改方法

```
    /**
     * 新增
     * @param ad
     */
    public void add(Ad ad) {
        adMapper.insert(ad);
        saveAdByPosition(ad.getPosition());//重新加载缓存
    }

    /**
     * 修改
     * @param ad
     */
    public void update(Ad ad) {
        //获取之前的广告位置
        String position =
adMapper.selectByPrimaryKey(ad.getId()).getPosition();
        adMapper.updateByPrimaryKeySelective(ad);
        saveAdToRedisByPosition(position);//更新
        if(!position.equals(ad.getPosition())){  //如果广告位置发生变化
            saveAdToRedisByPosition(ad.getPosition());
        }
    }

    /**
     *  删除
     * @param id
     */
    public void delete(Integer id) {
        String position = adMapper.selectByPrimaryKey(id).getPosition();
        saveAdByPosition(position);//重新加载缓存
        adMapper.deleteByPrimaryKey(id);
    }
```

5. 商品详细页价格缓存

5.1 需求分析

我们已经将商品的信息生成为静态页面，但是商品价格经常变动，如果每次价格变动后都对静态页重新生成会影响服务器性能。所以，对于商品价格，我们采用异步调用的方式来进行客户端渲染。

5.2 实现思路

（1）商品服务启动后加载全部价格数据到缓存。使用hash存储，skuID作为小KEY

（2）从缓存从查询商品价格，封装为controller，并设置可跨域调用

什么叫跨域？

当协议、子域名、主域名、端口号中任意一个不相同时，都算作不同域。不同域之间相互请求资源，就算作“跨域”。

JavaScript出于安全方面的考虑，不允许跨域调用其他页面的对象。那什么是跨域呢，简单地理解就是因为JavaScript同源策略的限制，a.com域名下的js无法操作b.com或是c.a.com域名下的对象。
当协议、子域名、主域名、端口号中任意一个不相同时，都算作不同域。不同域之间相互请求资源，就算作“跨域”。

现在我们要实现的查询商品价格缓存功能就存在跨域问题。后端controller在http://www.qingcheng.com ，商品详细页在http://item.qingcheng.com

如何解决跨域问题？我们使用CORS实现跨域。

CORS是一个W3C标准，全称是"跨域资源共享"（Cross-origin resource sharing）。它允许浏览器向跨源服务器，发出`XMLHttpRequest`请求，从而克服了AJAX只能同源使用的限制。使用非常简单，只需要在controller类上添加一个`@CrossOrigin`注解即可

（3）修改商品详细页模板，使用ajax读取价格，并进行客户端渲染。

5.3 代码实现

5.3.1 价格数据加载到缓存

（1）SkuService接口新增方法定义

```
    /**
     *  保存全部价格到缓存
     */
    public void saveAllPriceToRedis();
```

（2）SkuServiceImpl类新增方法

```
    public void saveAllPriceToRedis() {
        //检查缓存是否存在价格数据
        if(!redisTemplate.hasKey(CacheKey.SKU_PRICE)){
            System.out.println("加载全部价格");
            List<Sku> skuList = skuMapper.selectAll();
            for(Sku sku:skuList){
                if("1".equals(sku.getStatus())){
redisTemplate.boundHashOps(CacheKey.AD).put(sku.getId(),sku.getPrice());
                }
            }
        }else{
            System.out.println("已存在价格数据，没有全部加载");
        }
    }
```

（3）修改InitService类

```
@Component
public class InitService implements InitializingBean {

    @Autowired
    private CategoryService categoryService;

    @Autowired
    private SkuService skuService;

    public void afterPropertiesSet() throws Exception {
        System.out.println("---缓存预热---");
        categoryService.saveCategoryTreeToRedis();//加载商品分类导航缓存
        skuService.saveAllPriceToRedis();//加载价格数据
    }
}
```

5.3.2 查询价格缓存

后端代码：

（1）SkuService新增方法定义

```
public Integer findPrice(String id)
```

（2）SkuServiceImpl实现findPrice方法

```
    public Integer findPrice(String id) {
        //从缓存中查询
        return
(Integer)redisTemplate.boundHashOps(CacheKey.SKU_PRICE).get(id);
    }
```

（3）qingcheng_web_portal工程新增类

```
@RestController
@RequestMapping("/sku")
@CrossOrigin
public class SkuController {

    @Reference
    private SkuService skuService;

    @GetMapping("/price")
    public Integer price(String id){
        return skuService.findPrice(id);
    }
}
```

前端代码（修改模板）：

（1）将vue.js axios.js 放到我们html输出文件夹下的js文件夹中。

（2）修改qingcheng_web_portal工程的模板 item.html

```
<script src="js/vue.js"></script>
<script src="js/axios.js"></script>
<script th:inline="javascript">

    new Vue({
        el:'#app',
        data(){
            return {
                skuId:/*[[${sku.id}]]*/,
                price:0
            }
        },
        created(){
            //读取价格
            axios.get('http://localhost:9102/sku/price.do?
id='+this.skuId).then(response=>{
                this.price=(response.data/100).toFixed(2);
            })
        }

    });


</script>
```

th:inline　定义js脚本可以使用变量　js脚本的变量用`/*[[${　　}]]*/`渲染

（3）修改qingcheng_web_portal工程的模板 item.html ，添加`<div id="app">...</div>`，并将价格修改为vue表达式

```
{{price}}
```

5.3.3 更新价格缓存

（1）SkuService接口新增方法定义

```
    /**
     *  保存价格到缓存
     * @param skuId
     */
    public void savePriceToRedisById(String id,Integer price);
```

（2）SkuServiceImpl类新增方法

```
    public void savePriceToRedisById(String id,Integer price) {
        redisTemplate.boundHashOps(CacheKey.SKU_PRICE).put(id,price);
    }
```

（3）SpuServiceImpl类引入SkuService

```
@Autowired
private SkuService skuService;
```

（4）修改SpuServiceImpl类saveGoods方法，在SKU列表循环体中添加代码

```
skuService.savePriceToRedisById(sku.getId(),sku.getPrice());
```

5.3.4 删除价格缓存

删除商品时删除缓存中的价格，释放内存空间

（1）SkuService新增方法定义

```
    /**
     * 根据sku id 删除商品价格缓存
     * @param id
     */
    public void deletePriceFromRedis(String id);
```

（2）SkuServiceImpl新增方法实现

```
    public void deletePriceFromRedis(String id) {
        redisTemplate.boundHashOps(CacheKey.SKU_PRICE).delete(id);
    }
```

（3）修改SpuServiceImpl的delete方法，新增代码逻辑

```
        //删除缓存中的价格
        Map map=new HashMap();
        map.put("spuId",id);
        List<Sku> skuList = skuService.findList(map);
        for(Sku sku:skuList){
            skuService.deletePriceFromRedis(sku.getId());
        }
```