尚品汇商城复习

版本：V 1.0

商品详情模块

一、业务介绍

商品详情页，简单说就是以购物者的角度展现一个 sku 的详情信息。

这个页面不同于传统的 crud 的详情页，使用者并不是管理员，需要对信息进行查删改查，取而代之的是点击购买、放入购物车、切换颜色等等。

另外一个特点就是该页面的高访问量，虽然只是一个查询操作，但是由于频繁的访问所以我们必须对其性能进行最大程度的优化。

商品详情所需构建的数据如下：

1，Sku 基本信息------skuId

2，Sku 图片信息

3，Sku 所属分类信息

4，Spu 销售属性相关信息，Sku 对应的销售属性默认选中

5，商品切换的组合{127|128:34}

6，Sku 价格

7，商品介绍内容主体(海报)

8，查询规格属性

9、查询 spu 的评论

二、 使用缓存实现优化

虽然咱们实现了页面需要的功能，但是考虑到该页面是被用户高频访问的，所以性能必须进行尽可能的优化。

一般一个系统最大的性能瓶颈，就是数据库的 io 操作。从数据库入手也是调优性价比最高的切入点。

一般分为两个层面，一是提高数据库 sql 本身的性能，二是尽量避免直接查询数据库。

提高数据库本身的性能首先是优化 sql，包括：使用索引，减少不必要的大表关联次数，控制查询字段的行数和列数。另外当数据量巨大是可以考虑分库分表，以减轻单点压力。

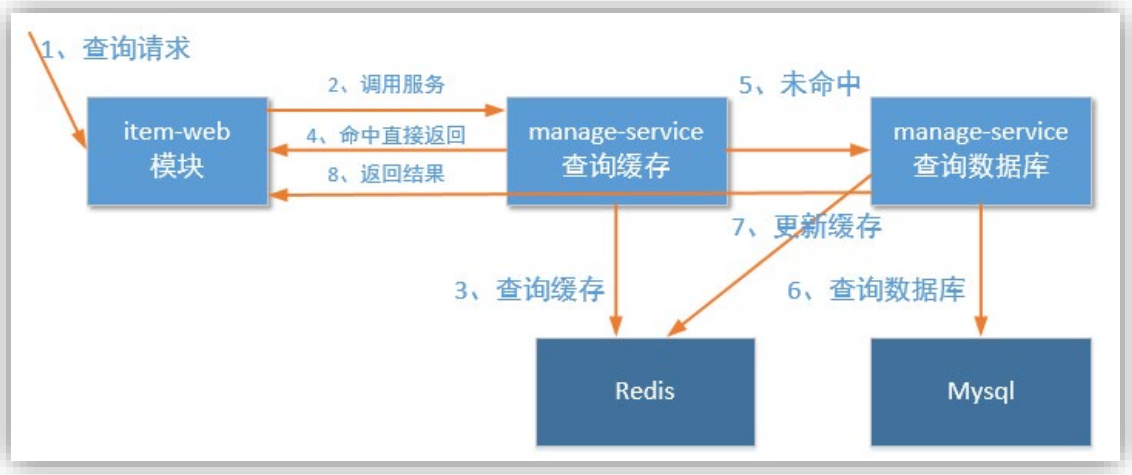
重点要讲的是另外一个层面：尽量避免直接查询数据库。

解决办法就是：缓存

缓存可以理解是数据库的一道保护伞，任何请求只要能在缓存中命中，都不会直接访问数据库。而缓存的处理性能是数据库 10-100 倍。

咱们就用 Redis 作为缓存系统进行优化。

结构图：



三、Redis 相关面试（重点）

1、简单介绍一个 redis？

非关系：数据和数据之间 没有强制的关联关系

NoSQL：去除 SQL 语句，使用命令的形式。

（1）redis 是一个 key-value 类型的非关系型数据库，基于内存也可持久化的数据库，相对于关系型数据库（数据主要存在硬盘中），性能高，因此我们一般用 redis 来做缓存使用；并且 redis 支持丰富的数据类型，比较容易解决各种问题

（2）Redis 支持 5 种基本数据类型 ，3 种特殊类型 bitmap

String 类型是最简单的类型，一个 key 对应一个 value； set get

Hash 类型中的 key 是 string 类型，value 又相当于一个 map（key-value）；

Hset hget

{key:[{filed:value},{ filed:value },{ filed:value },{ filed:value }……]}

List 类型是按照插入顺序的字符串链表（双向链表），主要命令是 LPUSH 和 RPUSH，能够支持反向查找和遍历；

Set 类型是用哈希表类型的字符串序列，没有顺序，集合成员是唯一的，没有重复数据。

zset（sorted set）类型和 set 类型基本是一致的，不同的是 zset 这种类型会给每个元素关联一个 **double** 类型的分数（score），这样就可以为成员排序，并且插入是有序的。

2、Redis 在你们项目中是怎么用的？

（1）商品详情中的数据放入缓存--string，分布式锁-----string；

（2）单点登录系统中也用到了 redis。因为我们是微服务系统，把用户信息存到 redis 中便于多系统之间统一获取-----string；

（3）我们项目中同时也将购物车的信息设计存储在 redis 中，用户未登录采用 UUID 作为 Key，value 是购物车对象；用户登录之后将商品添加到购物车后存储到 redis 中，key 是用户 id，value 是购物车对象；

使用 hash {userId 或者 userTempId:[{skuId:cartInfo},{ skuId:cartInfo },{ skuId:cartInfo }]}

（4）订单模块的，结算页中订单流水号 String 类型；

（5）秒杀中使用 list 类型控制库存；

（6）搜索中商品热度统计，使用 sorted set（Zset）；

Set 类型没有用，它可以去交集，共同关注、共同好友

还有一些其他的应用场景，主要就是用来作为缓存使用。

3、对 redis 的持久化了解不？

Redis 是内存型数据库，同时它也可以持久化到硬盘中，redis 的持久化方式有两种：

Redis.conf----核心配置

#Bind 127.0.0.1

（1）RDB（半持久化方式）：

按照配置不定期的通过异步的方式、快照的形式直接把内存中的数据持久化到磁盘的一个 dump.rdb 文件（二进制文件）中；

这种方式是 redis 默认的持久化方式，它在配置文件（redis.conf）中的格式是：save N M，表示的是在 N 秒之内发生 M 次修改，则 redis 抓快照到磁盘中；

优点：只包含一个文件，对于文件备份、灾难恢复而言，比较实用。因为我们可以轻松的将一个单独的文件转移到其他存储媒介上；性能最大化，因为对于这种半持

久化方式，使用的是写时拷贝技术，可以极大的避免服务进程执行 IO 操作；相对于 AOF 来说，如果数据集很大，RDB 的启动效率就会很高

缺点：如果想保证数据的高可用（最大限度的包装数据丢失），那么 RDB 这种半持久化方式不是一个很好的选择，因为系统一旦在持久化策略之前出现宕机现象，此前没有来得及持久化的数据将会产生丢失；rdb 是通过子进程来协助完成持久化的，因此当数据集较大的时候，我们就需要等待服务器停止几百毫秒甚至一秒；

（2）AOF（全持久化的方式）

把每一次数据变化都通过 write()函数将你所执行的命令追加到一个 appendonly.aof 文件里面；

Redis 默认是不支持这种全持久化方式的，需要将 no 改成 yes

实现文件刷新的三种方式：

```
# Redis supports three different modes:
#
# no: don't fsync, just let the OS flush the data when it wants. Faster.
# always: fsync after every write to the append only log . Slow, Safest.
# everysec: fsync only if one second passed since the last fsync. Compromise.
#
```

no:不会自动同步到磁盘上，需要依靠 OS（操作系统）进行刷新，效率快，但是安全性就比较差；

always:每提交一个命令都调用异步刷新到 aof 文件，非常慢，但是安全；

everysec:每秒钟都调用 fsync 刷新到 aof 文件中，很快，但是可能丢失一秒内的数据，推荐使用，兼顾了速度和安全；

如果 redis 数据一个不能丢 always everysec

优点：

数据安全性高

该机制对日志文件的写入操作采用的是 append 模式，因此在写入过程中即使出现宕机问题，也不会破坏日志文件中已经存在的内容；

缺点：

对于数量相同的数据集来说，aof 文件通常要比 rdb 文件大，因此 rdb 在恢复大数据集时的速度大于 AOF；

根据同步策略的不同，AOF 在运行效率上往往慢于 RDB，每秒同步策略的效率是比较高的，同步禁用策略的效率和 RDB 一样高效；

针对以上两种不同的持久化方式，如果缓存数据安全性要求比较高的话，用 aof 这种持久化方式（比如项目中的购物车）；如果对于大数据集要求效率高的话，就可以使用默认的。而且这两种持久化方式可以同时使用。

4、做过 redis 的集群吗？你们做集群的时候搭建了几台，都是怎么搭建的？

针对这类问题，我们首先考虑的是为什么要搭建集群？（这个需要针对我们的项目来说）Redis 的数据是存放在内存中的，这就意味着 redis 不适合存储大数据，大数据存储一般公司常用 hadoop 中的 Hbase 或者 MogoDB。因此 redis 主要用来处理高并发的，用我们的项目来说，电商项目如果并发大的话，一台单独的 redis 是不能足够支持我们的并发，这就需要我们扩展多台设备协同合作，即用到集群。

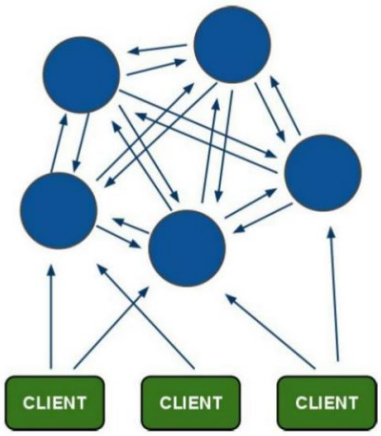
Redis 搭建集群的方式有多种， redis3.0 之后就支持 redis-cluster 集群，这种方式采用的是无中心结构，每个节点保存数据和整个集群的状态，每个节点都和其他所有节点连接。如果使用的话就用 redis-cluster 集群。

集群这块直接说是公司运维搭建的，小公司的话也有可能由我们自己搭建，开发环境我们也可以直接用单机版的。但是可以了解一下 redis 的集群版。搭建 redis 集群的时候，对于用到多少台服务器，每家公司都不一样，大家针对自己项目的大小去衡量。举个简单的例子：

我们项目中 redis 集群主要搭建了 6 台，3 主（为了保证 redis 的投票机制）3 从（高可用），每个主服务器都有一个从服务器，作为备份机。

【扩展】

1、架构图如下：



（1）所有的节点都通过 PING-PONG 机制彼此互相连接；

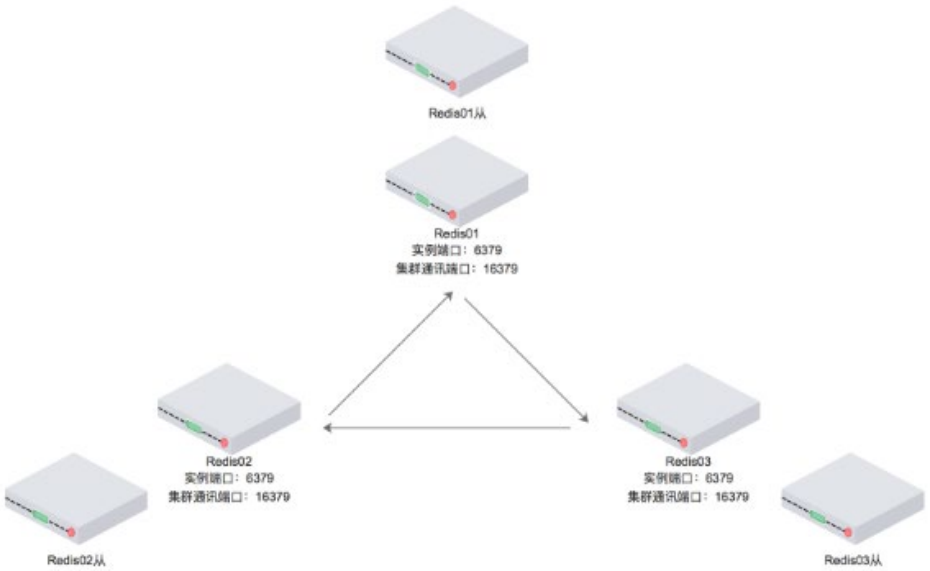
（2）每个节点的 fail 是通过集群中超过半数的节点检测失效时才生效；

（3）客户端与 redis 集群连接，只需要连接集群中的任何一个节点即可；

（4）Redis-cluster 把所有的物理节点映射到【0-16383】slot 上，负责维护

2、容错机制（投票机制）

（1）选举过程是集群中的所有 master 都参与，如果半数以上 master 节点与故障节点连接超过时间，则认为该节点故障，自动会触发故障转移操作；



图中描述的是六个 redis 实例构成的集群

6379 端口为客户端通讯端口

16379 端口为集群总线端口

集群内部划分为 16384 个数据分槽，分布在三个主 redis 中。

从 redis 中没有分槽，不会参与集群投票，也不会帮忙加快读取数据，仅仅作为主机的备份。

三个主节点中平均分布着 16384 数据分槽的三分之一，每个节点中不会存有有重复数据，仅仅有自己的从机帮忙冗余。

（2）集群不可用?

a:如果集群任意 master 挂掉，并且当前的 master 没有 slave，集群就会 fail;

b:如果集群超过半数以上 master 挂掉，无论是否有 slave，整个集群都会 fail;

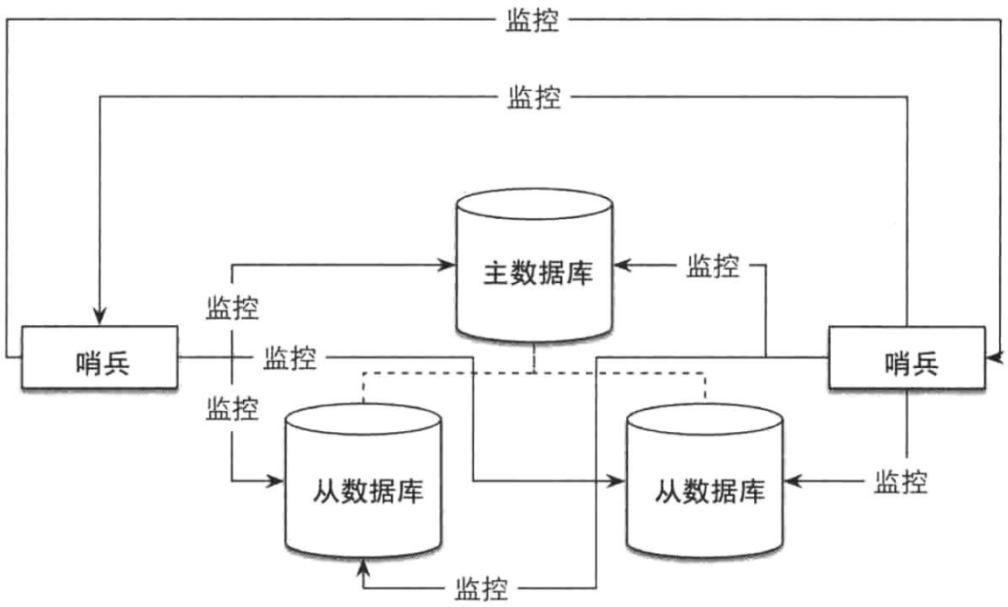
5、redis 的哨兵机制---2.6 版本

哨兵机制：

监控：监控主数据库和从数据库是否正常运行，监控所有 redis 节点;

提醒：当被监控的某个 redis 出现问题的时候，哨兵可以通过 API 向管理员或者其他应用程序发送通知;

自动故障迁移：主数据库出现故障时，可以自动将从数据库转化为主数据库，实现自动切换;



具体的配置步骤面试中可以说参考的网上的文档。要注意的是，如果 master 主服务器设置了密码，记得在哨兵的配置文件（sentinel.conf）里面也要配置访问密码

四、缓存存在的问题与解决（重点中的重点）

缓存最常见的 3 个问题：

缓存雪崩：

Redis 不起作用，大量的访问直接查数据库

1、 所有 redis 服务不可用，保证高可用（集群）

2、 大量的 key 集群失效了（过期了）。

解决：在添加缓存的时候 过期时间设置随机。

缓存穿透：

查询一个数据，缓存里没有，去查数据库，然后数据库也没有。

这个数据根本就不存在，导致后续多次访问都访问数据库了。

解决：简单版本解决 使用 没有值的对象，放入缓存，过期时间较短，无用数据。

负杂的就是 布隆过滤器 1970 年 由布隆这个人提出的。判断一个数据在一个集合中是否存在。

Redis 的 bitmap 可以实现。比较麻烦

Redisson 给封装了，实现了。

缓存击穿：

一个**热点**的 key，缓存中正好过期了，这时有大量的访问 直接到数据库了。

解决：

分布式锁。

叫分布式这三字的 ，有两个，一个是分布式锁，分布式事务

分布式锁：

1、 Redis 的 setNX +过期时间（防止死锁） ----只有 key 不存在的时候才能写进去。

只有一个线程 set 进去了，其他线程就 set 不了了。

如果抛异常 try catch finally{释放锁，删除 key}

这个版本有锁的误删问题。

2、 setNX+过期时间+lua 脚本（删除的原子性，使用了 uuid，这个 UUID 是锁的值）

3、 redisson 的 lock 锁 lock.trylock() lock.unlock()

4、 最终版 自定义注解+ AOP+redisson 分布式锁 抽取，加以复用。

项目刚上线的时候 缓存里没数据，这时 1000 个商品，每个商品有 100 个人同时访问。

缓存预热：把所有需要缓存的数据 提前加入到缓存中。缓存预热的时候 key 过期时间设置成随机的。

1、缓存穿透

是指查询一个不存在的数据，由于缓存无法命中，将去查询数据库，但是数据库也无此记录，并且出于容错考虑，我们没有将这次查询的 null 写入缓存，这将导致这个不存在的数据每次请求都要到存储层去查询，失去了缓存的意义。在流量大时，可能 DB 就挂掉了，要是有人利用不存在的 key 频繁攻击我们的应用，这就是漏洞。

解决：空结果也进行缓存，但它的过期时间会很短，最长不超过五分钟。

布隆过滤器

2、缓存雪崩

是指在我们设置缓存时采用了相同的过期时间，导致缓存在某一时刻同时失效，请求全部转发到 DB，DB 瞬时压力过重雪崩。

解决：原有的失效时间基础上增加一个随机值，比如 1-5 分钟随机，这样每一个缓存的过期时间的重复率就会降低，就很难引发集体失效的事件。

3、缓存击穿

是指对于一些设置了过期时间的 key，如果这些 key 可能会在某些时间点被超高并发地访问，是一种非常“热点”的数据。这个时候，需要考虑一个问题：如果这个 key 在大量请求同时进来之前正好失效，那么所有对这个 key 的数据查询都落到 db，我们称为缓存击穿。

与缓存雪崩的区别:

1. 击穿是一个热点 key 失效

2. 雪崩是很多 key 集体失效

解决方案:

随着业务发展的需要，原单体单机部署的系统被演化成分布式集群系统后，由于分布式系统多线程、多进程并且分布在不同机器上，这将使原单机部署情况下的并发控制锁策略失效，单纯的 Java API 并不能提供分布式锁的能力。为了解决这个问题就需要一种跨 JVM 的互斥机制来控制共享资源的访问，这就是分布式锁要解决的问题！

使用**分布式锁**，采用 redis 的 KEY 过期时间实现

命令+key 的过期时间

Redis:命令 setNX+key 的过期时间

```
# set skuid:1:info “OK” NX PX 10000
```

EX second ： 设置键的过期时间为 second 秒。

PX millisecond ： 设置键的过期时间为 millisecond 毫秒。

NX ： 只在键(key)不存在时，才对键(key)进行设置操作。

XX ： 只在键(key)已经存在时，才对键(key)进行设置操作。

Redis SET 命令用于设置给定 key 的值，如果 key 已经存在其他值，SET 就会覆盖，且无视类型。

问题：删除操作缺乏原子性。

场景：

1. index1 执行删除时，查询到的 lock 值确实和 uuid 相等
2. index1 执行删除前，lock 刚好过期时间已到，被 redis 自动释放
3. index2 获取了 lock
4. index1 执行删除，此时会把 index2 的 lock 删除

解决：使用 LUA 脚本保证删除的原子性

使用 redisson 解决分布式锁

redisson ： 工具

官方文档地址：https://github.com/redisson/redisson/wiki

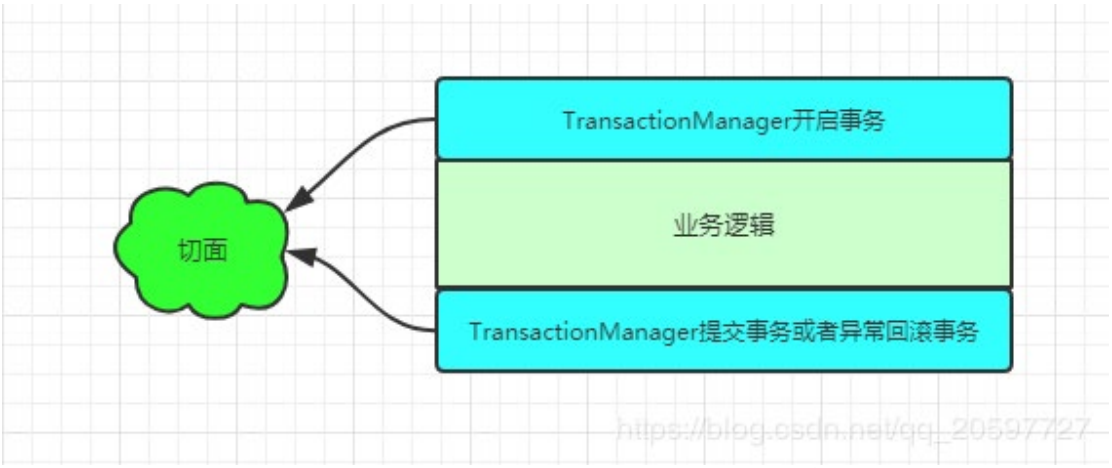
连接文档：https://github.com/redisson/redisson

```
RLock lock = redisson.getLock("anyLock");
// 最常使用
lock.lock();
// 加锁以后 10 秒钟自动解锁
// 无需调用 unlock 方法手动解锁
lock.lock(10, TimeUnit.SECONDS);
// 尝试加锁，最多等待 100 秒，上锁以后 10 秒自动解锁
boolean res = lock.tryLock(100, 10, TimeUnit.SECONDS);
if (res) {
   try {
     ...
   } finally {
       lock.unlock();
   }
}
```

4、分布式锁 + AOP 实现缓存

随着业务中缓存及分布式锁的加入，业务代码变的复杂起来，除了需要考虑业务逻辑本身，还要考虑缓存及分布式锁的问题，增加了程序员的工作量及

开发难度。而缓存的玩法套路特别类似于事务，而声明式事务就是用了 aop 的思想实现的。



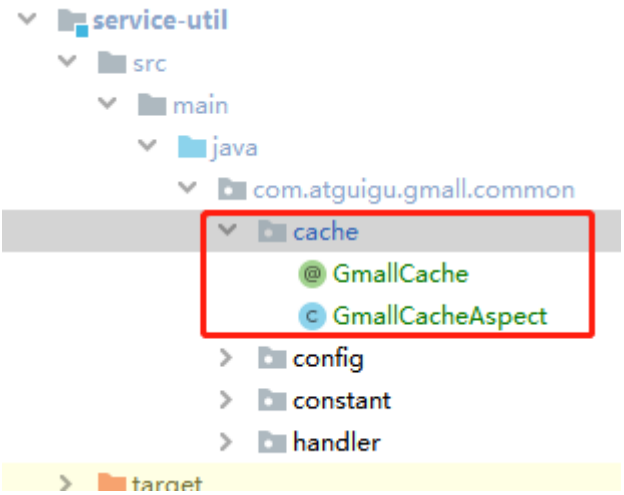
以 @Transactional 注解为植入点的切点，这样才能知道@Transactional 注解标注的方法需要被代理。

@Transactional 注解的切面逻辑类似于@Around

模拟事务，缓存可以这样实现：

1. 自定义缓存注解@GmallCache（类似于事务@Transactional）

2. 编写切面类，使用环绕通知实现缓存的逻辑封装



定义一个注解

```
package com.atguigu.gmall.common.cache;

import java.lang.annotation.*;

@Target({ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface GmallCache {
```

```
    /**
     * 缓存key的前缀
   * @return
     */
    String prefix() default "cache";
}
```

定义一个切面类加强注解

```
package com.atguigu.gmall.common.cache;



@Component
@Aspect
public class GmallCacheAspect {

    @Autowired
    private RedisTemplate redisTemplate;

    @Autowired
    private RedissonClient redissonClient;

    /**
     * 1.返回值object
     * 2.参数proceedingJoinPoint
     * 3.抛出异常Throwable
     * 4.proceedingJoinPoint.proceed(args)执行业务方法
   */

@Around("@annotation(com.atguigu.gmall.common.cache.GmallCache)")
    public Object cacheAroundAdvice(ProceedingJoinPoint point)
throws Throwable {

        Object result = null;
        // 获取连接点签名
    MethodSignature signature = (MethodSignature)
point.getSignature();
        // 获取连接点的GmallCache注解信息
    GmallCache gmallCache =
signature.getMethod().getAnnotation(GmallCache.class);
        // 获取缓存的前缀
    String prefix = gmallCache.prefix();

        // 组装成key
        String key = prefix +
```

```
Arrays.asList(point.getArgs()).toString();

        // 1. 查询缓存
    result = this.cacheHit(signature, key);

        if (result != null) {
            return result;
        }

        // 初始化分布式锁
    RLock lock = this.redissonClient.getLock("gmallCache");
        // 防止缓存穿透 加锁
    lock.lock();

        // 再次检查内存是否有，因为高并发下，可能在加锁这段时间内，已有其他线程放入缓存
    result = this.cacheHit(signature, key);
        if (result != null) {
            lock.unlock();
            return result;
        }

        // 2. 执行查询的业务逻辑从数据库查询
    result = point.proceed(point.getArgs());
        // 并把结果放入缓存
    this.redisTemplate.opsForValue().set(key, JSONObject.toJSONString(result));

        // 释放锁
    lock.unlock();

        return result;
    }

    /**
     * 查询缓存的方法
  *
     * @param signature
     * @param key
     * @return
     */
    private Object cacheHit(MethodSignature signature, String key) {
        // 1. 查询缓存
    String cache = (String)redisTemplate.opsForValue().get(key);
        if (StringUtils.isNotBlank(cache)) {
            // 有，则反序列化，直接返回
      Class returnType = signature.getReturnType(); // 获取方法返回类
```

```
型
        // 不能使用parseArray<cache, T>，因为不知道List<T>中的泛型
        return JSONObject.parseObject(cache, returnType);
            }
            return null;
        }
}
```

使用注解完成缓存

```
@GmallCache(prefix = RedisConst.SKUKEY_PREFIX)
@Override
public SkuInfo getSkuInfo(Long skuId) {

    return getSkuInfoDB(skuId);
}
```

五、使用异步线程优化商品详情

问题：查询商品详情页的逻辑非常复杂，数据的获取都需要远程调用，必然需要花费更多的时间。

假如商品详情页的每个查询，需要如下标注的时间才能完成

```
// 1. 获取 sku 的基本信息      0.5s

// 2. 获取 sku 的图片信息      0.5s

// 3. 获取 spu 的所有销售属性          1s

// 4. sku 价格  1.5s

//5、增加热度


可能 调用评论接口

...
```

那么，用户需要 4.5s 后才能看到商品详情页的内容。很显然是不能接受的。

如果有多个线程同时完成这 4 步操作，也许只需要 1.5s 即可完成响应。

使用 CompletableFuture 实现异步线程优化商品详情

```
@Service
public class ItemServiceImpl implements ItemService {
    @Autowired
    private ProductFeignClient productFeignClient;
    @Autowired
    private ThreadPoolExecutor threadPoolExecutor;
    @Override
    public Map<String, Object> getBySkuId(Long skuId) {
        Map<String, Object> result = new HashMap<>();
        // 通过skuId 查询skuInfo
        CompletableFuture<SkuInfo>       skuCompletableFuture       =
CompletableFuture.supplyAsync(() -> {
            SkuInfo skuInfo = productFeignClient.getSkuInfo(skuId);
            // 保存skuInfo
            result.put("skuInfo", skuInfo);
            return skuInfo;
        }, threadPoolExecutor);
        // 销售属性-销售属性值回显并锁定
    CompletableFuture<Void>       spuSaleAttrCompletableFuture       =
skuCompletableFuture.thenAcceptAsync(skuInfo -> {
            List<SpuSaleAttr>             spuSaleAttrList             =
productFeignClient.getSpuSaleAttrListCheckBySku(skuInfo.getId(),
skuInfo.getSpuId());
            // 保存数据
      result.put("spuSaleAttrList", spuSaleAttrList);
        }, threadPoolExecutor);
        //根据spuId 查询map 集合属性
    // 销售属性-销售属性值回显并锁定
    CompletableFuture<Void>     skuValueIdsMapCompletableFuture     =
skuCompletableFuture.thenAcceptAsync(skuInfo -> {
            Map                     skuValueIdsMap                  =
productFeignClient.getSkuValueIdsMap(skuInfo.getSpuId());
            String                    valuesSkuJson                 =
JSON.toJSONString(skuValueIdsMap);
            // 保存valuesSkuJson
            result.put("valuesSkuJson", valuesSkuJson);
        }, threadPoolExecutor);
       //获取商品最新价格
    CompletableFuture<Void>        skuPriceCompletableFuture        =
CompletableFuture.runAsync(() -> {
            BigDecimal                    skuPrice                  =
productFeignClient.getSkuPrice(skuId);
            result.put("price", skuPrice);
        }, threadPoolExecutor);
        //获取分类信息
    CompletableFuture<Void>      categoryViewCompletableFuture      =
skuCompletableFuture.thenAcceptAsync(skuInfo -> {
            BaseCategoryView              categoryView              =
```

```
productFeignClient.getCategoryView(skuInfo.getCategory3Id());
            //分类信息
      result.put("categoryView", categoryView);
        }, threadPoolExecutor);
        CompletableFuture.allOf(skuCompletableFuture,
spuSaleAttrCompletableFuture,
skuValueIdsMapCompletableFuture,skuPriceCompletableFuture,
categoryViewCompletableFuture).join();
        return result;
    }
}
```

```
package com.atguigu.gmall.item.config;

@Configuration

public class ThreadPoolConfig {
    @Bean
    public ThreadPoolExecutor threadPoolExecutor(){
        /**
         * 核心线程数
    * 拥有最多线程数
    * 表示空闲线程的存活时间
    * 存活时间单位
    * 用于缓存任务的阻塞队列
    * 省略:
    *  threadFactory: 指定创建线程的工厂
    *  handler: 表示当 workQueue 已满，且池中的线程数达到 maximumPoolSize 时，线
程池拒绝添加新任务时采取的策略。
    */
        return    new    ThreadPoolExecutor(50,500,30,    TimeUnit.SECONDS,new
ArrayBlockingQueue<>(10000));
    }
}
```