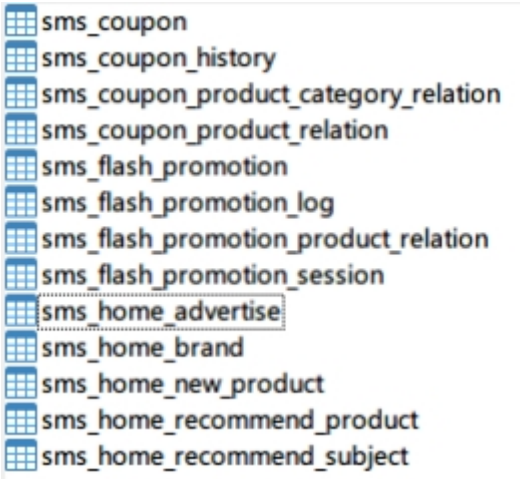
电商缓存方案设计

商城项目的缓存实践

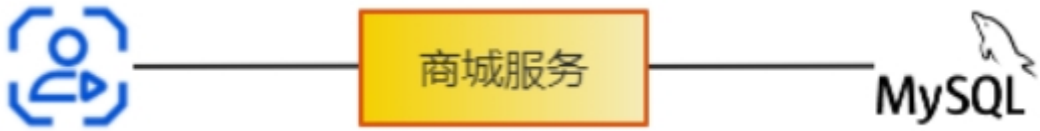
基础实现

商城项目中有多个地方使用到了缓存来提升性能，最显著的就是tulingmall-portal，商城的首页入口服务。

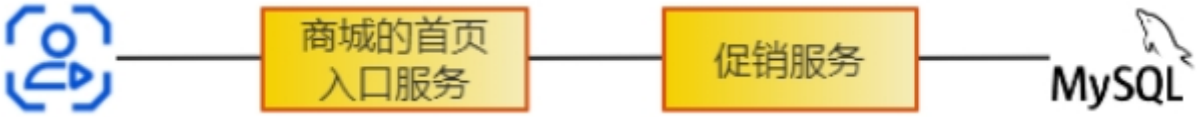
既然是商城的首页入口，那么必然的它就是整个商城系统访问最多的的一个部分，大体有推荐品牌、人气推荐、新品推荐、轮播广告以及秒杀活动这几个部分，而这几个部分都属于促销信息，数据则存储在



在商城系统的早期（四期及以前）版本，这些数据是直接从数据表中读取然后显示在商城的首页上，访问路径为：



在我们现在的商城系统，做了微服务拆分，上述的数据表全部归属于tulingmall-promotion促销管理服务负责访问路径为：



那就意味着不管是早期版本还是微服务版本，不做优化的话，首页其实是无法应对高并发的。

在前面的课程我们说过，如果任何人看到的内容都是一样的（推荐系统要紧密结合大数据，不在本课程的考虑范围），也就是说，对后端服务来说，任何人的查询请求和返回的数据都是一样的。在这种情况下，缓存的命中率非常高，几乎所有的请求都可以命中缓存，很自然，首页上的这些展示很符合这些特性，所以我们想到用缓存来应对首页的高并发。

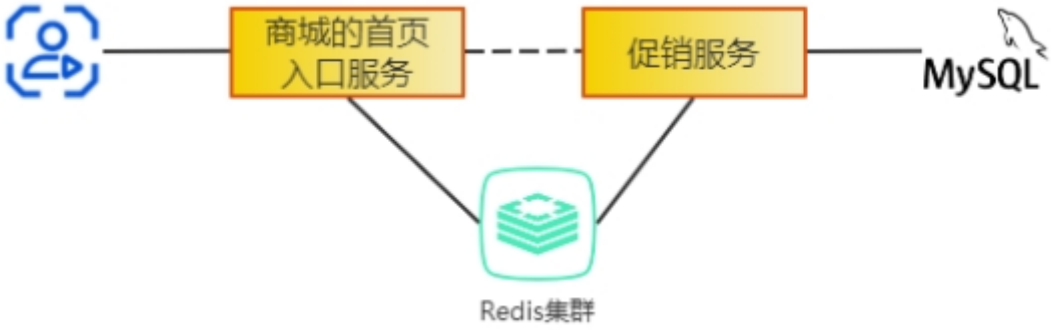
为了保护数据库应对高并发，我们考虑首先将促销信息放入缓存，所以在tulingmall-promotion的HomePromotionServiceImpl中，这些数据在获取时（以新品推荐为例）：

```
PageHelper.startPage( pageNum: 0,ConstantPromotion.HOME_RECOMMEND_PAGESIZE, orderBy:
SmsHomeNewProductExample example = new SmsHomeNewProductExample();
example.or().andRecommendStatusEqualTo(ConstantPromotion.HOME_PRODUCT_RECOMMEND_YES);
List<Long> newProductIds = smsHomeNewProductMapper.selectProductIdByExample(example);
newProducts = pmsProductClientApi.getProductBatch(newProductIds);
redisOpsExtUtil.putListAllRight(newProductKey,newProducts);
```

可以看到，我们从数据库中获取后，也往Redis缓存集群中存放了一份，很自然的，当首页获取新品推荐时，自然就可以Redis集群中获得，以下代码在tulingmall-portal的HomeServiceImpl

```
/*从远程(Redis或者对应微服务)获取推荐内容*/
4 usages    Mark
public HomeContentResult getFromRemote(){
    List<PmsBrand> recommendBrandList = null;
    List<SmsHomeAdvertise> smsHomeAdvertises = null;
    List<PmsProduct> newProducts  = null;
    List<PmsProduct> recommendProducts  = null;
    HomeContentResult result = null;
    /*从redis获取*/
    if(promotionRedisKey.isAllowRemoteCache()){
        recommendBrandList = redisOpsUtil.getListAll(promotionRedisKey.getBrandKey
        smsHomeAdvertises = redisOpsUtil.getListAll(promotionRedisKey.getHomeAdver
        newProducts = redisOpsUtil.getListAll(promotionRedisKey.getNewProductKey()
        recommendProducts = redisOpsUtil.getListAll(promotionRedisKey.getRecProduc
    }
    /*redis没有则从微服务中获取*/
    if(CollectionUtil.isEmpty(recommendBrandList)
            ||CollectionUtil.isEmpty(smsHomeAdvertises)
            ||CollectionUtil.isEmpty(newProducts)
            ||CollectionUtil.isEmpty(recommendProducts)) {
        result = promotionFeignApi.content( getType: 0).getData();
```

很自然的，访问路径变为：



通过这种方式，我们以高性能的Redis取代MySQL，并且缩短了整个访问路径，提升了首页服务的性能。

在前面我们说过，缓存一定是离用户越近越好，依据这个原则，首页还有优化的空间，从上面的访问路径可以看到，首页服务需要到Redis集群中获得数据用以展示，能不能将缓存的数据再提前呢？于是我们在首页服务内引入了应用级缓存Caffeine。

TIPS：Caffeine基于Google的Guava Cache，提供一个性能卓越的本地缓存(local cache) 实现，也是SpringBoot内置的本地缓存实现，有资料表明Caffeine性能是Guava Cache的6倍

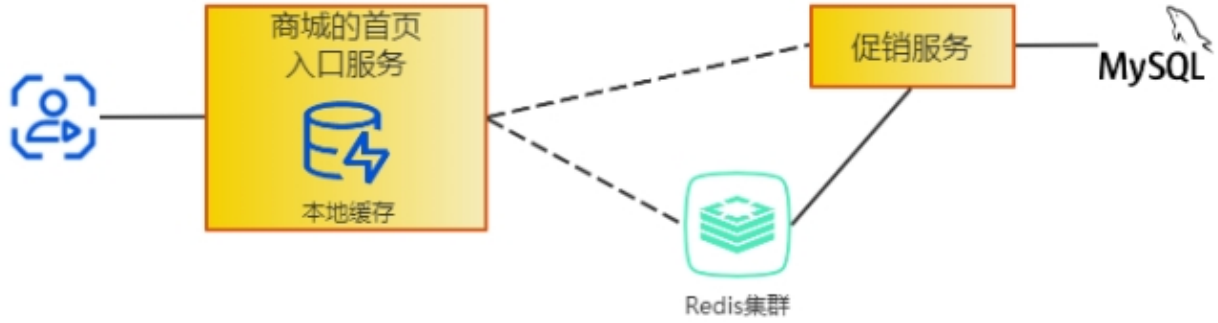
```
public class CaffeineCacheConfig {

    Mark
    @Bean(name = "promotion")
    public Cache<String, HomeContentResult> promotionCache() {
        int rnd = ThreadLocalRandom.current().nextInt( bound: 10);
        return Caffeine.newBuilder()
                // 设置最后一次写入经过固定时间过期
                .expireAfterWrite( duration: 30 + rnd, TimeUnit.MINUTES)
                // 初始的缓存空间大小
                .initialCapacity(20)
                // 缓存的最大条数
                .maximumSize(100)
                .build();
    }
```

于是在tulingmall-portal的HomeServiceImpl中，对于查询请求我们先从本地缓存中获取，未获取到再从远程获取

```
public HomeContentResult recommendContent(){
    /*品牌和产品在本地缓存中统一处理，有则视为同有，无则视为同无*/
    final String brandKey = promotionRedisKey.getBrandKey();
    final boolean allowLocalCache = promotionRedisKey.isAllowLocalC
    /*先从本地缓存中获取推荐内容*/
    HomeContentResult result = allowLocalCache ?
            promotionCache.getIfPresent(brandKey) : null;
    if(result == null){
        result = allowLocalCache ?
                promotionCacheBak.getIfPresent(brandKey) : null;
    }
    /*本地缓存中没有*/
    if(result == null){
        log.warn("从本地缓存中获取推荐品牌和商品失败，可能出错或禁
        result = getFromRemote();
        if(null != result) {
            promotionCache.put(brandKey,result);
            promotionCacheBak.put(brandKey,result);
        }
    }
```

访问路径自然就变为下图所示，并通过这种方式进一步提升了首页的访问性能。



促销数据先从tulingmall-portal的本地Caffeine缓存获得，如果没有，再从远程获得，获取步骤首先从Redis集群获取，Redis没有，则从tulingmall-promotion中获取，tulingmall-promotio则会从数据库中获取并写入Redis集群。此时tulingmall-portal获得远程的应答后，将数据写入本地的Caffeine缓存。

到了这一步，我们在首页中引入缓存的工作并没有完成，还有几个缓存使用的问题需要解决。

缓存预热

首页服务如果出现重启会导致缓存中的首页数据丢失，导致查询请求直接访问数据库，所以在首页服务中有专门的缓存预热机制，在tulingmall-portal启动时从远程获得相关的数据写入本地的Caffeine缓存

```
public class PreheatCache implements CommandLineRunner {

    1 usage
    @Autowired
    private HomeService homeService;

    Mark
    @Override
    public void run(String... args) throws Exception {
        for(String str : args) {
            log.info("系统启动命令行参数: {}",str);
        }
        homeService.preheatCache();
    }

}
```

```
try {
    if(promotionRedisKey.isAllowLocalCache()){
        final String brandKey = promotionRedisKey.getBrandKey();
        HomeContentResult result = getFromRemote();
        promotionCache.put(brandKey,result);
        promotionCacheBak.put(brandKey,result);
        log.info("promotionCache 数据缓存预热完成");
    }
} catch (Exception e) {
    log.error("promotionCache 数据缓存预热失败：",e);
}
```

在tulingmall-promotion中也存在同样的缓存预热机制，负责将数据从MySQL数据写入到Redis集群中

```
public class PreheatCache implements CommandLineRunner {

    1 usage
    @Autowired
    private HomePromotionService homePromotionService;

    Mark
    @Override
    public void run(String... args) throws Exception {
        for(String str : args) {
            log.info("系统启动命令行参数: {}",str);
        }
        homePromotionService.content(ConstantPromotion.HOME_GET_TYPE_ALL);
    }
```

```
public HomeContentResult content(int getType) {
    HomeContentResult result = new HomeContentResult();
    if(ConstantPromotion.HOME_GET_TYPE_ALL == getType
            ||ConstantPromotion.HOME_GET_TYPE_BARND == getType){
        //获取推荐品牌
        getRecommendBrand(result);
    }
    if(ConstantPromotion.HOME_GET_TYPE_ALL == getType
            ||ConstantPromotion.HOME_GET_TYPE_NEW == getType){
        getRecommendProducts(result);
    }
    if(ConstantPromotion.HOME_GET_TYPE_ALL == getType
            ||ConstantPromotion.HOME_GET_TYPE_HOT == getType){
        getHotProducts(result);
    }
    if(ConstantPromotion.HOME_GET_TYPE_ALL == getType
            ||ConstantPromotion.HOME_GET_TYPE_AD == getType){
        //获取首页广告
        result.setAdvertiseList(getHomeAdvertiseList());
    }
    return result;
}
```

数据一致性

必然的，推荐品牌、人气推荐、新品推荐、轮播广告以及秒杀活动等促销信息一定存在着变化，当数据库中更新后，就和缓存中的数据不一致了，需要我们更新缓存。

对于tulingmall-portal的本地Caffeine缓存，我们设置了过期时间30分钟

```
return Caffeine.newBuilder()
        // 设置最后一次写入经过固定时间过期
        .expireAfterWrite( duration: 30 + rnd, TimeUnit.MINUTES)
        // 初始的缓存空间大小
        .initialCapacity(20)
        // 缓存的最大条数
        .maximumSize(100)
        .build();
```

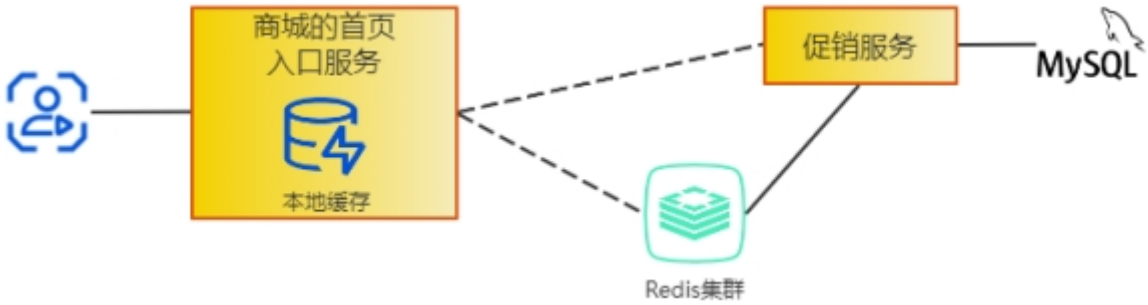
并在RefreshPromotionCache中以后台任务的形式异步的刷新缓存，每分钟检查一次本地Caffeine缓存是否已无效，无效则刷新缓存

```java
@Async
@Scheduled(initialDelay=5000*60,fixedDelay = 1000*60)
public void refreshCache(){
    if(promotionRedisKey.isAllowLocalCache()){
        log.info("检查本地缓存[promotionCache] 是否需要刷新...");
        final String brandKey = promotionRedisKey.getBrandKey();
        if(null == promotionCache.getIfPresent(brandKey)||null == promotionCacheBa
            log.info("本地缓存[promotionCache] 需要刷新");
            HomeContentResult result = homeService.getFromRemote();
            if(null != result){
                if(null == promotionCache.getIfPresent(brandKey)) {
                    promotionCache.put(brandKey,result);
                    log.info("刷新本地缓存[promotionCache] 成功");
                }
                promotionCacheBak.put(brandKey,result);
                log.info("刷新本地缓存[promotionCacheBak] 成功");
            }else{
                log.warn("从远程获得[promotionCache] 数据失败");
            }
        }
    }
}
```

而对于Redis集群中的数据，则是利用Canal监测数据库的更新，然后删除缓存中的对应部分，具体实现在tulingmall-canal数据同步程序的PromotionData中。Redis数据的再载入自然由tulingmall-promotion负责。

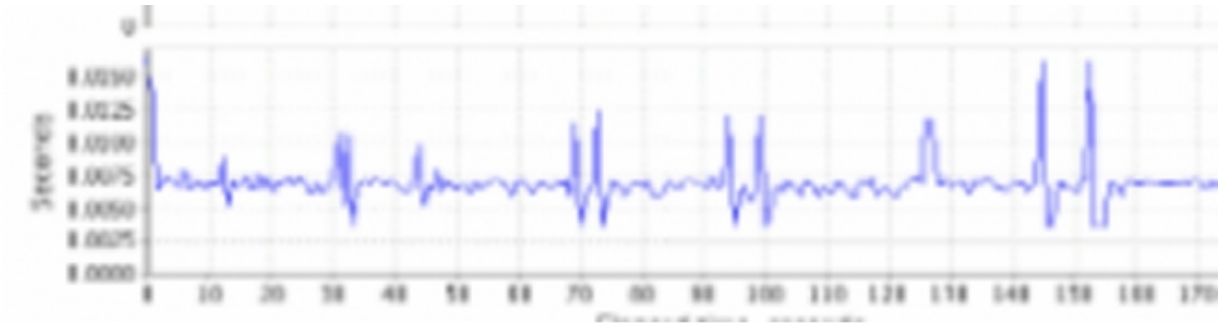
双缓存

从前面的促销数据的获得路径我们知道



促销数据先从tulingmall-portal的本地Caffeine缓存获得，如果没有，再从远程获得。为了数据的一致性，本地Caffeine设置了过期时间，Redis集群中的数据也会在数据变动后被删除。

这就会造成一种情况，当本地Caffeine缓存中已经失效，这个时候就需要去远程获取，最好的情况下可以从Redis集群中获得，最坏的情况下需要从数据库中获得。于是在远程获取的情况下，用户的访问链路变长，单次耗时变长，整体会出现类似下图很明显的有规律毛刺现象



如何避免这种情况呢？于是我们引入了双缓存机制：

```
@Bean(name = "promotion")
public Cache<String, HomeContentResult> promotionCache() {
    int rnd = ThreadLocalRandom.current().nextInt( bound: 10);
    return Caffeine.newBuilder()
            // 设置最后一次写入经过固定时间过期
            .expireAfterWrite( duration: 30 + rnd, TimeUnit.MINUTES)
            // 初始的缓存空间大小
            .initialCapacity(20)
            // 缓存的最大条数
            .maximumSize(100)
            .build();
}

/*以双缓存的形式提升首页的访问性能，这个备份缓存其实运行过程中会永不过期
 * 可以作为首页的降级和兜底方案 * */
 Mark
@Bean(name = "promotionBak")
public Cache<String, HomeContentResult> promotionCacheBak() {
    int rnd = ThreadLocalRandom.current().nextInt( bound: 10);
    return Caffeine.newBuilder()
            // 设置最后一次访问经过固定时间过期
            .expireAfterAccess( duration: 41 + rnd, TimeUnit.MINUTES)
            // 初始的缓存空间大小
            .initialCapacity(20)
            // 缓存的最大条数
            .maximumSize(100)
            .build();
}
```

也就是促销数据在本地的缓存我们保存了两份，并且将备份缓存作为降级和兜底方案。

在首页获得促销数据的过程中，我们会在两份本地缓存中获取

```
/*先从本地缓存中获取推荐内容*/
HomeContentResult result = allowLocalCache ?
        promotionCache.getIfPresent(brandKey) : null;
if(result == null){
    result = allowLocalCache ?
            promotionCacheBak.getIfPresent(brandKey) : null;
}
```

只有两份本地缓存都没有，才会到远程获得。

在数据的过期机制上可以看到，我们使用了不同的策略，正式缓存是最后一次写入后经过固定时间过期，备份缓存是设置最后一次访问后经过固定时间过期。这就意味着备份缓存中内容不管是读写后，实际过期时间都会后延，正式缓存中的数据在被读取后，实际过期时间不会后延。

在本地缓存的异步刷新机制上，正式缓存只有无效才会被重新写入，备份缓存无论是否无效都会重新写入，一则可以保证备份缓存中的数据不至于真的永久无法过期而太旧，二则使备份缓存的过期时间不管用户是否访问首页都可以不断后延。

同时tulingmall-portal中也提供了专门的缓存管理接口CacheManagerController，方便进行强制本地缓存失效和手动刷新本地缓存。

其他

秒杀的活动信息在首页的是单独处理的，但是处理的思路是一致的，只有细微的地方有所不同，这里不再赘述。

其实首页的这些信息还可以使用固定的图片或者Http连接作为本地缓存、Redis集群和tulingmall-promotion微服务均失效的最终的降级和兜底方案，这个可以自行实现。

同时我们系统在缓存的使用上，还有可以改进的空间，比如注册用户的布隆过滤器化以应对缓存穿透，商品信息的缓存化等等。

> 有道云笔记链接：https://note.youdao.com/s/5DcuodmN