Please visit website: http://cxyroad.com

MySQL里有2000万条数据，但是Redis中只存20万的数据，如何保证redis中的数据都是热点数据?
==========

引言
--

在当今互联网领域，尤其在大型电商平台如淘宝这样的复杂分布式系统中，数据的高效管理和快速访问至关重要。面对数以千万计的商品、交易记录以及其他各类业务数据，如何在MySQL等传统关系型数据库之外，借助内存数据库Redis的力量，对部分高频访问数据进行高效的缓存处理，是提升整个系统性能的关键一环。

比如淘宝，京东，拼多多等电商系统每日处理的订单量级庞大，其数据库中存储的商品、用户信息及相关交易数据可达数千万条。为了降低数据库查询的压力，加速数据读取，Redis常被用于搭建二级缓存系统，以容纳部分最为活跃的“热点数据”。然而，在资源有限的情况下，如何确保仅有的20万条缓存数据精准匹配到系统中的热点数据，避免频繁的冷数据替换热数据导致的缓存失效，这就涉及到了一套精密的数据管理策略和缓存淘汰机制的设计。

本文将围绕这一实战场景展开讨论：在MySQL拥有2000万条数据的前提下，如何确保Redis仅缓存的20万条数据全都是系统中的热点数据，从而最大程度上发挥缓存的优势，提高系统的响应速度和并发能力，进而提升用户的购物体验和服务质量。通过对Redis内部机制的深入理解以及对业务场景的精细分析，我们将揭示一套综合运用各种技术手段来确保Redis中热点数据准确有效的管理方案。

![image.png](https://p3-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/01e06e532b34452696006901181710b5~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=1574&h=996&s=143190&e=png&a=1&b=ffffff ”null”)

image.png

技术背景

在探讨如何确保Redis中存储的20万数据均为热点数据之前，首先需要明确MySQL与Redis在实际业务环境中的互补关系以及Redis自身的内存管理和数据淘汰机制。

MySQL与Redis的关系及应用场景

MySQL作为一种成熟的关系型数据库管理系统，适用于存储大量持久化且具有复杂关系的数据，其强大的事务处理能力和安全性保障了数据的一致性和完整性。但在大规模并发环境下，尤其是对那些读多写少、访问频次极高的热点数据，直接从MySQL中读取可能会成为系统性能瓶颈。

Redis则是一种高性能的内存键值数据库，以其极快的速度和灵活的数据结构著称。在淘宝这类大型电商平台中，Redis主要用于缓存频繁访问的数据，例如热门商品信息、用户购物车、会话状态等，以此减轻主数据库的压力，提高响应速度，增强系统的可扩展性和容错性。

> 对于Redis高性能原理，请参考：[京东二面：Redis为什么快？我说Redis是纯内存操作的，然后他对我笑了笑。](http://cxyroad.com/”https://mp.weixin.qq.com/s?__biz=MzkyNzYzMTY0MA==&mid=2247484198&idx=1&sn=32b0992f1b7d9862e2c983f9cb2e633f&chksm=c2245351f553da472f19d57c7d7dfa56c58e8ea4a9e814fc2d4a8fadf72afeadae4b0b0bec35#rd”) 对于Redis的使用的业务场景，请参考：[美团一面：项目中使用过Redis吗？我说用Redis做缓存。他对我哦了一声](http://cxyroad.com/”https://mp.weixin.qq.com/s?__biz=MzkyNzYzMTY0MA==&mid=2247484244&idx=1&sn=b1c1f81d63b5334260500fec82bb3f53&chksm=c2245323f553da350a87c63701e89a6556599814c93ca7d85c41424d07778c0ed721c6424c92#rd”)

Redis内存管理和数据淘汰机制简介

Redis的所有数据都存储在内存中，这意味着它的容量相较于磁盘存储更为有限。为了解决内存容量不足的问题，Redis提供了多种数据淘汰策略。其中，与保证热点数据密切相关的是LFU（Least Frequently Used）策略，它能够根据数据对象的访问频次，将访问次数最少（即最不常用）的数据淘汰出内存，以便为新的数据腾出空间。

> 对于Redis高性能的一方面原因就是Redis高效的管理内存，具体请参考：[京东二面：Redis为什么快？我说Redis是纯内存操作的，然后他对我笑了笑。](http://cxyroad.com/

”https://mp.weixin.qq.com/s?__biz=MzkyNzYzMTY0MA==&mid=2247484198&idx=1&sn=32b0992f1b7d9862e2c983f9cb2e633f&chksm=c2245351f553da472f19d57c7d7dfa56c58e8ea4a9e814fc2d4a8fadf72afeadae4b0b0bec35#rd”）

此外，Redis允许用户根据自身需求选择不同的淘汰策略，例如“volatile-lfu”只针对设置了过期时间的key采用LFU算法，“allkeys-lfu”则对所有key都执行LFU淘汰规则。

热点数据定义及其识别方法

热点数据是指在一定时间内访问频率极高、对系统性能影响重大的数据集。在电商平台中，这可能表现为热销商品详情、活动页面信息、用户高频查询的搜索关键词等。识别热点数据主要依赖于对业务日志、请求统计和系统性能监控工具的分析，通过收集和分析用户行为数据，发现并量化哪些数据是系统访问的热点，以便有针对性地将它们缓存至Redis中。

实现方案

在实际应用中，确保Redis中存储的数据为热点数据，我们可以从以下几个方案考虑实现。

LFU淘汰策略

Redis中的LFU（Least Frequently Used）淘汰策略是一种基于访问频率的内存管理机制。当Redis实例的内存使用量达到预先设定的最大内存限制（由`maxmemory`配置项指定）时，LFU策略会根据数据对象的访问频次，将访问次数最少（即最不常用）的数据淘汰出内存，以便为新的数据腾出空间。

LFU算法的核心思想是通过跟踪每个键的访问频率来决定哪些键应当优先被淘汰。具体实现上，Redis并非实时精确地计算每个键的访问频率，而是采用了近似的LFU方法，它为每个键维护了一个访问计数器（counter）。每当某个键被访问时，它的计数器就会递增。随着时间推移，Redis会依据这些计数器的值来决定淘汰哪些键。

在Redis 4.0及其后续版本中，LFU策略可以通过设置`maxmemory-policy`配置项为`allkeys-lfu`或`volatile-lfu`来启用。其中：

* • `allkeys-lfu`：适用于所有键，无论它们是否有过期时间，都会基于访问频率淘汰键。
* • `volatile-lfu`：仅针对设置了过期时间（TTL）的键，按照访问频率淘汰键。

Redis实现了自己的LFU算法变体，它使用了一个基于访问计数和老化时间的组合策略来更好地适应实际情况。这意味着不仅考虑访问次数，还会考虑到键的访问频率随时间的变化，防止长期未访问但曾经很热门的键占据大量内存空间而不被淘汰。在实现上，Redis使用了一种称为“频率跳表（frequency sketch）”的数据结构来存储键的访问频率，允许快速查找和更新计数器。为了避免长期未访问但计数器较高的键永久保留，Redis会在一段时间后降低键的访问计数，模拟访问频率随时间衰减的效果。

在Redis中使用LFU淘汰策略，在配置文件`redis.conf`中找到`maxmemory-policy`选项，将其设置为LFU相关策略之一：

```
maxmemory-policy allkeys-lfu # 对所有键启用LFU淘汰策略
# 或者
maxmemory-policy volatile-lfu # 对有过期时间的键启用LFU淘汰策略
```

确保你也设置了Redis的最大内存使用量（`maxmemory`），只有当内存到达这个上限时，才会触发淘汰策略：

```
maxmemory <size_in_bytes> # 指定Redis可以使用的最大内存大小
```

LFU策略旨在尽可能让那些近期最不活跃的数据优先被淘汰，以此保持缓存中的数据相对活跃度更高，提高缓存命中率，从而提升系统的整体性能。（这也是我们面试中需要回答出来的答案）

LRU淘汰策略

Redis中的LRU（Least Recently Used）淘汰策略是一种用于在内存不足时自动删除最近最少使用的数据以回收内存空间的方法。尽管Redis没有完全精确地

实现LRU算法（因为这在O(1)时间内实现成本较高），但Redis确实提供了一种近似LRU的行为。

当我们配置了最大内存限制，如果内存超出这个限制时，Redis会选择性地删除一些键值对来腾出空间。Redis提供了几种不同的淘汰策略，其中之一就是`volatile-lru`和`allkeys-lru`，这两种都试图模拟LRU行为。

* • **volatile-lru**：仅针对设置了过期时间（TTL）的键，按照最近最少使用的原则来删除键。
* • **allkeys-lru**：不论键是否设置过期时间，都会根据最近最少使用的原则来删除键。

Redis实现LRU的方式并不是真正意义上的双向链表加引用计数这样的完整LRU结构，因为每个键值对的插入、删除和访问都需要维持这样的数据结构会带来额外的开销。所以Redis实现LRU会采取以下方式进行：

1. 1. Redis内部为每个键值对维护了一个“空转时间”（idle time）的字段，它是在Redis实例启动后最后一次被访问或修改的时间戳。
2. 2. 当内存达到阈值并触发淘汰时，Redis不会遍历整个键空间找出绝对意义上的最近最少使用的键，而是随机抽取一批键检查它们的空转时间，然后删除这批键中最久未被访问的那个。 Redis在大多数情况下能较好地模拟LRU效果，有助于保持活跃数据在内存中，减少因频繁换入换出带来的性能损失。

内存淘汰策略通常是在Redis服务器端的配置文件（如`redis.conf`）中设置，而不是在应用中配置。你需要在Redis服务器端的配置中设置`maxmemory-policy`参数为`allkeys-lru`。（同LFU策略）

使用Redis的LRU淘汰策略实现热点数据的方式，简单易行，能较好地应对大部分情况下的热点数据问题。但是若访问模式复杂或数据访问分布不均匀，单纯的LRU策略可能不够精准，不能确保绝对的热点数据留存。

结合访问频率设定过期时间

在实际应用中，除了依赖Redis的淘汰策略外，还可以结合业务逻辑，根据数据的访问频率动态设置Key的过期时间。例如，当某个Key被频繁访问时，延长其在Redis中的有效期，反之则缩短。

```
```

```
@Autowired
private RedisTemplate<String, Object> redisTemplate;

public void updateKeyTTL(String key, int ttlInSeconds) {
    redisTemplate.expire(key, ttlInSeconds, TimeUnit.SECONDS);
}

// 示例调用，当检测到某个数据访问增多时，增加其缓存过期时间
public void markAsHotSpot(String key) {
    updateKeyTTL(key, 3600); // 将热点数据缓存时间延长至1小时
}
```

这种方式灵活性强，可根据实际访问情况动态调整缓存策略。但是需要在应用程序中进行较多定制开发，以捕捉并响应数据访问的变化。

基于时间窗口的缓存淘汰策略

在给定的时间窗口（如过去1小时、一天等）内，对每个数据项的访问情况进行实时跟踪和记录，可以使用计数器或其他数据结构统计每条数据的访问次数。到达时间窗口边界时，计算每个数据项在该窗口内的访问频率，这可以是绝对访问次数、相对访问速率或者其他反映访问热度的指标。根据预先设定的阈值，将访问次数超过阈值的数据项加入Redis缓存，或者将其缓存时间延长以确保其能在缓存中停留更久。而对于访问次数低于阈值的数据项，要么从缓存中移除，要么缩短其缓存有效期，使其更容易被后续淘汰策略处理。

```
@Service
public class TimeWindowCacheEvictionService {

    @Autowired
    private RedisTemplate<String, Object> redisTemplate;

    private final Map<String, AtomicInteger> accessCounts = new Concur
rentHashMap<>();

    // 时间窗口长度（例如，1小时）
    private static final long TIME_WINDOW_MILLIS = TimeUnit.HOURS.to
Millis(1);

    @Scheduled(fixedRate = TIME_WINDOW_MILLIS)
    public void evictBasedOnFrequency() {
        accessCounts.entrySet().forEach(entry -> {
```

```
            int accessCount = entry.getValue().get();
            if (accessCount > THRESHOLD) { // 假设THRESHOLD是访问次数阈值
                // 将数据存入或更新到Redis缓存，并设置较长的过期时间
                redisTemplate.opsForValue().set(entry.getKey(), getDataFromDB(entry.getKey()), CACHE_EXPIRATION_TIME, TimeUnit.MINUTES);
            } else if (redisTemplate.hasKey(entry.getKey())) {
                // 访问次数低，从缓存中移除或缩短过期时间
                redisTemplate.delete(entry.getKey());
            }
        });

        // 清零访问计数器，准备下一个时间窗口
        accessCounts.clear();
    }

    public void trackDataAccess(String dataId) {
        accessCounts.computeIfAbsent(dataId, k -> new AtomicInteger()).incrementAndGet();
    }
}
```

> 关于@Scheduled是Springboot中实现定时任务的一种方式，对于其他几种方式，请参考：[玩转SpringBoot：SpringBoot的几种定时任务实现方式](http://cxyroad.com/”https://mp.weixin.qq.com/s?__biz=MzkyNzYzMTY0MA==&mid=2247484100&idx=1&sn=17f9a506495eb273baa47d2d2efc16d1&chksm=c22452b3f553dba504ce7e133931848986696e6709a09427f741134107285c8f1550ecf03eddc#rd”)

通过这种方法，系统能够基于实际访问情况动态调整缓存内容，确保Redis缓存中存放的总是具有一定热度的数据。当然，这种方法需要与实际业务场景紧密结合，并结合其他缓存策略共同作用，以实现最优效果。同时，需要注意此种策略可能带来的额外计算和存储成本。

手动缓存控制

针对已识别的热点数据，可以通过监听数据库变更或业务逻辑触发器主动将数据更新到Redis中。例如，当商品销量剧增变为热点商品时，立即更新Redis缓存。

这种方式可以确保热点数据及时更新，提高了缓存命中率。

利用数据结构优化

使用Sorted Set等数据结构可以进一步精细化热点数据管理。例如，记录每个商品最近的访问的活跃时间，并据此决定缓存哪些商品数据。

```
// 商品访问活跃时更新其在Redis中的排序
String goodsActivityKey = "goods_activity";
redisTemplate.opsForZSet().add(goodsActivityKey, sku, System.currentTimeMillis());

// 定时清除较早的非热点商品数据
@Scheduled(cron = "0 0 3 * * ?") // 每天凌晨3点清理前一天的数据
public void cleanInactiveUsers() {
    long yesterdayTimestamp = System.currentTimeMillis() - TimeUnit.DAYS.toMillis(1);
    redisTemplate.opsForZSet().removeRangeByScore(goodsActivityKey, 0, yesterdayTimestamp);
}
```

这种方式能够充分利用Redis内建的数据结构优势，实现复杂的数据淘汰逻辑。

实际业务中实践方案

在例如淘宝这样庞大的电商生态系统中，面对MySQL中海量的业务数据和Redis有限的内存空间，我们采用了多元化的策略以确保缓存的20万数据是真正的热点数据。

LFU策略的运用

自Redis 4.0起，我们可以通过配置Redis淘汰策略为近似的LFU（`volatile-lfu` 或 `allkeys-lfu`），使得Redis能够自动根据数据访问频率进行淘汰决策。LFU策略基于数据的访问次数，使得访问越频繁的数据越不容易被淘汰，从而更好地保持了热点数据在缓存中的存在。

访问频率动态调整

除了依赖Redis内置的LFU淘汰策略，我们还可以实现应用层面的访问频率追踪和响应式缓存管理。例如，每当商品被用户访问时，系统会更新该商品在Redis中的访问次数，同时根据访问频率动态调整缓存过期时间，确保访问频率高的商品在缓存中的生存期得到延长。

```
@Service
public class ProductService {

    @Autowired
    private RedisTemplate<String, Product> redisTemplate;

    public void updateProductViewCount(String productId) {
        // 更新产品访问次数
        redisTemplate.opsForValue().increment("product:view_count:" + productId);

        // 根据访问次数调整缓存过期时间
        Long viewCount = redisTemplate.opsForValue().get("product:view_count:" + productId);
        if (viewCount > THRESHOLD_VIEW_COUNT) {
            redisTemplate.expire("product:info:" + productId, LONGER_CACHE_EXPIRATION, TimeUnit.MINUTES);
        }
    }
}
```

数据结构优化

我们还可以利用Redis丰富的数据结构，如有序集合（Sorted Sets）和哈希（Hashes），来实现商品热度排行、用户行为分析等功能。例如，通过Sorted Set存储商品的浏览量，自动按照浏览量高低进行排序，并淘汰访问量低的商品缓存。

```
// 更新商品浏览量并同步到Redis有序集合
public void updateProductRanking(String productId, long newViewCount)
```

```
{
    redisTemplate.opsForZSet().add("product_ranking", productId, newViewCount);
    // 自动淘汰浏览量低的商品缓存
    redisTemplate.opsForZSet().removeRange("product_ranking", 0, -TOP_RANKED_PRODUCT_COUNT - 1);
}
```

总结

本文详细阐述了在电商平台例如淘宝及其他类似场景下，如何结合LFU策略与访问频率调整，优化Redis中20万热点数据的管理。通过配置Redis近似的LFU淘汰策略，结合应用层面对访问频率的实时追踪与响应式调整，以及利用多样化的Redis数据结构如有序集合和哈希表，成功实现了热点数据的精确缓存与淘汰。

通过电商平台的一些实际业务实践证明，这种综合策略可以有效提升缓存命中率，降低数据库访问压力，确保缓存资源始终服务于访问最频繁的数据。未来随着数据挖掘与分析技术的进步，以及Redis或其他内存数据库功能的拓展，预计将进一步细化和完善热点数据的识别与管理机制。例如，探索更具前瞻性的预测性缓存策略，或是结合机器学习模型对用户行为进行深度分析，以更精准地预判和存储未来的热点数据。

本文已收录于我的个人博客：[码农Academy的博客，专注分享Java技术干货，包括Java基础、Spring Boot、Spring Cloud、Mysql、Redis、Elasticsearch、中间件、架构设计、面试题、程序员攻略等](http://cxyroad.com/ "https://www.coderacademy.online/")

原文链接: https://juejin.cn/post/7357546247849508902