Redis诞生于2009年全称是Remote Dictionary Server,远程词典服务器，是一个基于内存的键值型NoSQL数据库。

特征：

·键值型，value支持多种不同数据结构，功能丰富

·单线程，每个命令具备原子性，

·低延迟，速度快（基于内存、IO多路复用、良好的编码）

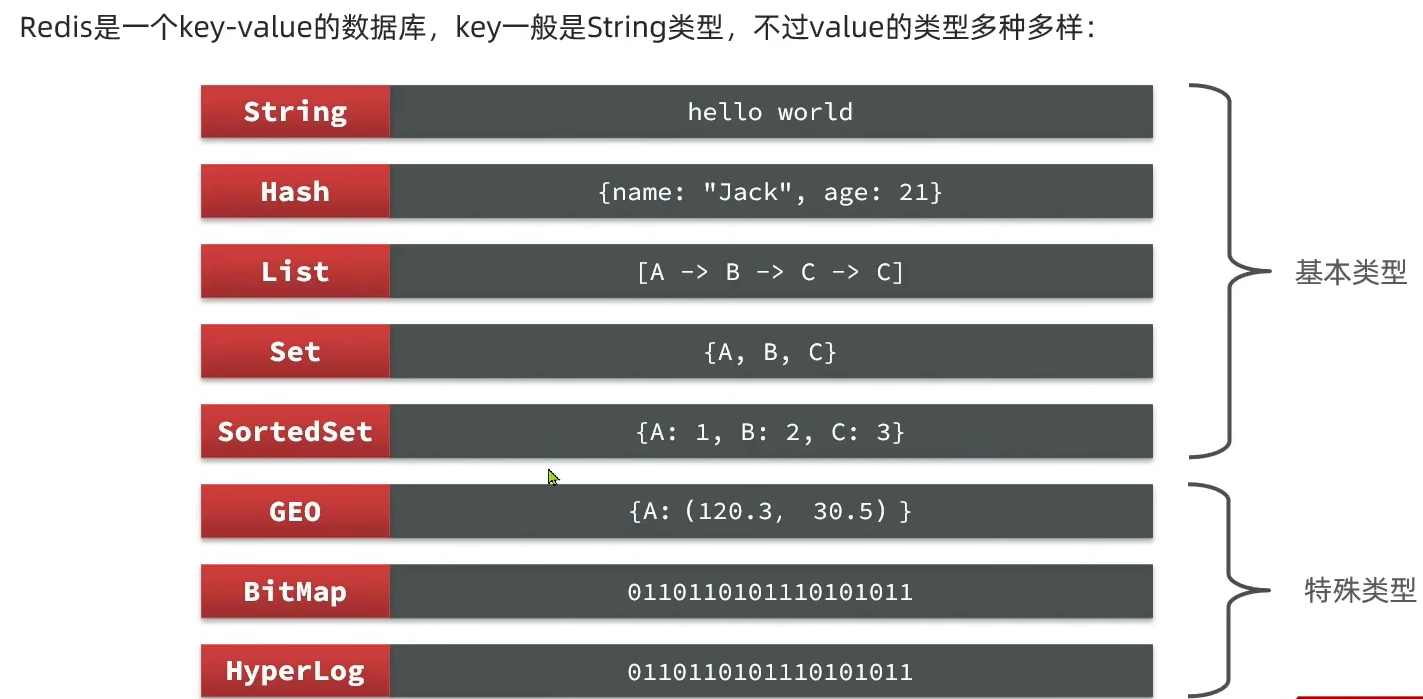
·支持数据持久化

·支持主从集群、分片集群

·支持多语言客户端

Redis数据结构介绍

Redis是一个key-value的数据库，key一般是String，value的类型多种多样



Redis通用命令

通用指令是部分数据类型得，都可以使用的指令。常见的有：

KEYS：查看符合模板得所有key

DEL:删除一个指定的key

EXISTS:判断key是否存在

EXPIRE:给一个key设置有效期，有效期到期时该key会被自动删除

TTL:查看一个key得剩余有效期

String类型

String类型，就是字符串类型，是Redis中最简单的存储类型

其value是字符串，不过根据字符串得格式不同，又可以分成三类

String：普通字符串

Int：整数类型，可以做自增，自减操作

Float：浮点类型，可以做自增，自减操作

不管是哪种格式，底层都是字节数组形式存储，只不过是编码方式不同，字符串类型的最大空间不能超过512m

String类型得常见命令：

SET：添加或者修改已经存在得一个String类型得键值对

GET:根据key获取String类型得value

MSET：批量添加多个String类型得键值对

MGET：根据多个key获取多个String类型的value

INCR:让一个整形的key自增1

INCRBY:让一个整形的key自增并指定步长，例如：INCRBY num 2 让key为num得值增2

INCRBYFLOAT:让一个浮点型得数字自增并指定步长

SETNX:添加一个String类型的键值对，前提是这个key不存在，否则不执行

SETEX:添加一个String类型的键值对，并指定有效期

Key得结构

Redis得key允许有多个单词形成层级结构，多个单词之间用“：”隔开

Hash类型

Hash类型，也叫散列，其value是一个无序字典，类似于java中的HashMap结构

Hash结构可以将对象中的每个字段独立存储，可以针对单个字段CRUD

List类型

Redis中的list类型和java中的LinkedList类似，可以看作是一个双向链表，既支持正向检索也支持反向检索

特征也和LinkedList类似

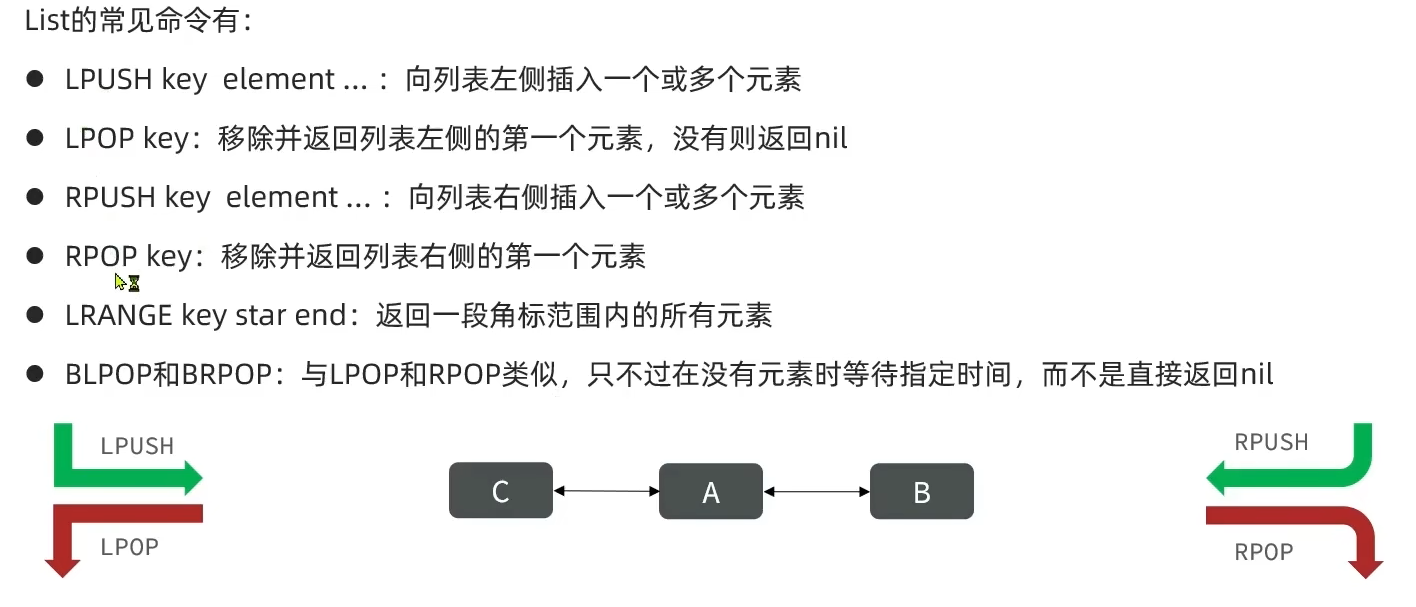
有序

元素可重复

插入和删除快

查询速度一般

List的常见命令：



Set类型

Redis的set结构和java中的hashset类似，可以看作是一个value为null的hashMap。因为也是一个hash表，因此具备与hashset类似的特征

无序

元素不重复

查找快

支持交集并集差集等

SAKK key member ：像元素添加一个或多个元素

SREM key member ：移除set中的指定元素

SCARD key ：返回set中元素的个数

SISMEMBER key member：判断一个元素是否存在于set中

SMEMBERS:获取set中的所有元素

SINTER key1 key2 ：求key1和key2的交集

SDIFF key1 key2：求key1和key2的差集

SUNION key1 key2：求key1和key2的并集

SortedSet类型

Redis的sortedset是一个可排序的set集合，与java中的treeset有些类似，但底层数据结构差别很大，sortedset中的每一个元素都带一个score属性，可以基于score属性对元素排序，底层的实现是一个调表（SkipList）加hash表。

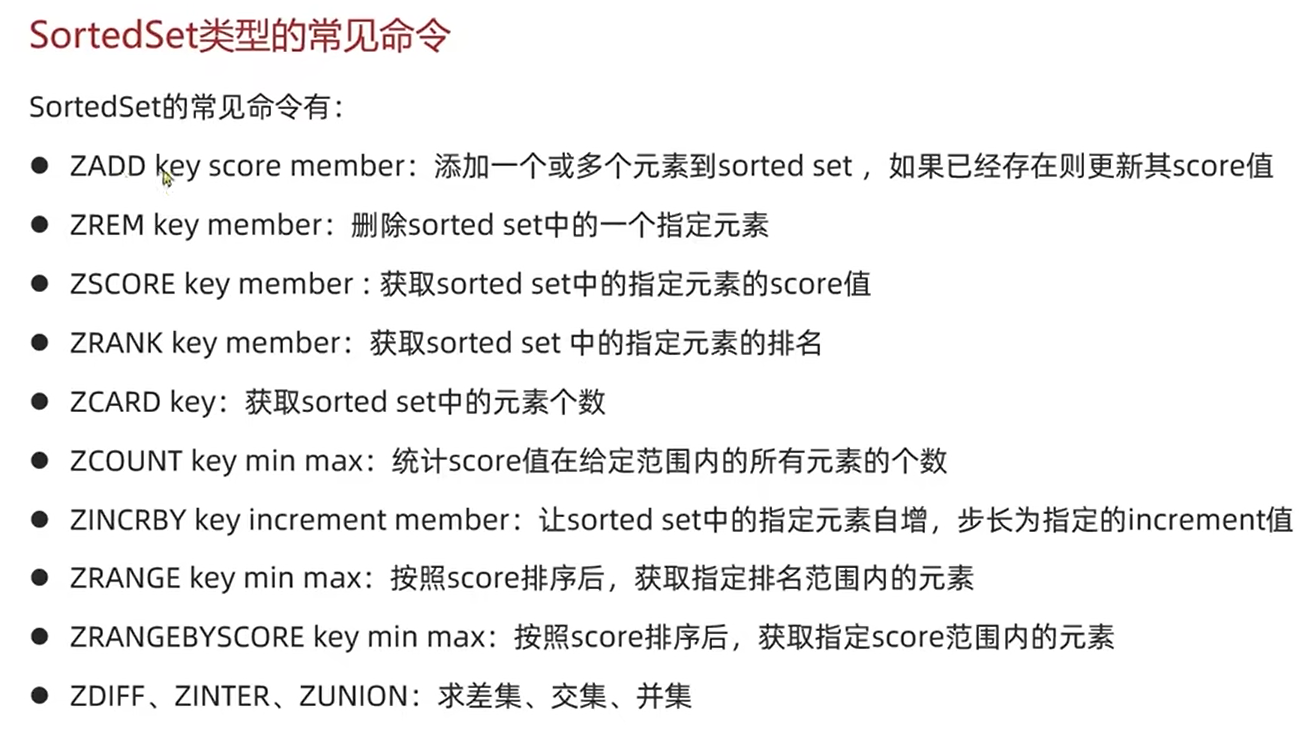
SortedSet具备下列特性：

可排序

元素不重复

查询速度快

因为SortedSet的可排序特性，经常被用来实现排行榜这样的功能



SpringDataRedis

SpringData是Spring中数据操作的模块，包含对各种数据库的继承，其中对redis的集成的模块就叫SpringDataRedis

提供了对不同Redis客户端的整合

提供了RedisTemplate统一API来操作Redis

支持Redis的发布订阅模型

支持Redis哨兵和集群

支持基于Lettuce的响应式编程

支持基于Json，JDK，字符串，Spring对象的数据序列化及反序列化

支持基于Redis的JDKCollection

RedisTemplate可以接受任意Object作为值写入redis，只不过写入前会把Object序列化为字节形式，默认是采用JDK序列化（ObjectOutPutStream）

缺点：可读性差，占用内存

所以我们手写一个RedisTemplate工厂当作一个bean注入，设置key和value的序列化格式

尽管JSON的序列化方式可以满足需求，但是依然存在问题：为了在反序列化时知道对象的类型，JSON序列化器会将类的class类型写入json，存入redis，占用内存

为了节省内存，我们不会使用json序列化器来处理value，而是统一使用String序列化器，要求只能存储String类型的key和value。当需要存储java对象时，手动完成对象的序列化和反序列化

Spring默认提供了一个SpringRedisTemplate类，他的key和value的序列化方式默认就是String方式，省去了我们自定义RedisTemplate的过程

缓存就是数据交换的缓冲区称为Cache,是存储数据的临时地方，一般读写性能比较高

缓存的作用：

降低后端负载

提高读写效率，降低响应时间

缓存的成本：

数据一致性的成本

代码维护成本

运维成本

|  |  |  |  |
| --- | --- | --- | --- |
|  | 内存淘汰 | 超时剔除 | 主动更新 |
| 说明 | 不用自己维护，利用redis的内存淘汰机制，当内存不足时自动淘汰部分数据，下次查询时更新数据 | 给缓存数据添加TTL时间，到期后自动删除缓存，下次查询时更新缓存 | 编写业务逻辑，在修改数据同时，更新缓存 |
| 一致性 | 差 | 一般 | 好 |
| 维护成本 | 无 | 低 | 高 |

主动更新策略

1. Cache Aside Pattern 由缓存的调用者，在更新数据库的同时更新缓存
2. Read/Write Through Pattern 缓存与数据库整合为一个服务，由服务来维护一致性，调用者调用该服务，无需关心缓存一致性问题
3. Write Behind Caching Pattern 调用者只操作数据库，由其他线程异步的将缓存数据持久化到数据库，保持最终一致性

操作缓存和数据库时有三个问题需要考虑

1. 删除缓存还是更新缓存

更新缓存：每次更新数据库都更新缓存，无效写操作较多

删除缓存：更新数据库时让缓存失效，查询时在更新缓存

1. 如何保证缓存与数据库的操作的同时成功和失败

单体系统，将缓存与数据库操作在一个事务

分布式系统，利用TCC等分布式事务方案

1. 先操作缓存还是先操作数据库

先删除缓存，在操作数据库

先操作数据库，再删除缓存 相对来说更安全

缓存更新策略的最佳实践

1. 低一致性：使用redis自带的缓存淘汰机制
2. 高一致性：主动更新，并以超时剔除作为兜底方案

读操作：缓存命中则返回，缓存未命中则查询数据库并写入

写操作：先写数据库再删，要确保数据库和缓存的原子性

缓存穿透

缓存穿透是指客户端请求的数据再缓存中和数据库中都不存在，这样缓存永远不会生效，这些请求都会打到数据库

解决方式：

缓存空对象： 第一次请求数据库不存在，再redis中缓存null

优点：实现简单，维护方便

缺点：

额外的内存消耗 优化设置TTL（短时间）

可能造成短期的不一致

布隆过滤：

优点：内存占用较少，没有多余key

缺点：实现复杂

存在误判可能

缓存雪崩

缓存雪崩是指同一时间段大量的缓存key同时失效或者Redis服务宕机，导致大量请求到达数据库，带来巨大压力

解决方案：

给不同的key的TTL添加随机值

利用Redis集群提高服务的可用性

给缓存业务添加降级限流策略

给业务添加多级缓存

缓存击穿

缓存击穿问题，也叫热点key问题，就是一个被高并发访问并且缓存重建业务较复杂的key突然失效，无数的请求就会在瞬间给数据库带来巨大的冲击。

常见的解决方案有两种：

互斥锁：

当线程1查询缓存未命中，获取互斥锁，查询数据库并更新缓存，线程2查询缓存，未命中，获取锁失败，则重试查询缓存和获取锁，当线程一写入缓存结束，线程二则缓存命中。

逻辑过期：

不设置key的TTL，则不会过期

查询缓存，发现逻辑时间已过期，获取互斥锁成功，，开启一个新线程，查询数据库重建缓存数据，写入缓存，重置逻辑过期时间，释放锁，则原来的线程返回过期数据，当另外一个线程查询缓存，发现逻辑时间过期，获取锁失败，则直接返回过期数据

全局ID生成器

全局ID生成器是一种在分布式系统下用来生成全局唯一ID得工具，一般满足以下特点：

唯一性

高可用

高性能

递增性

安全性

ID得组成部分

符号位：1bit 永远是0，表示正数

时间戳：31bit，以秒为单位，可以使用69年

序列号：32bit，秒内得计数器，支持每秒产生2得32次方个订单

超卖问题就是典型的多线程安全问题，针对这一问题得常见解决方案就是加锁：

**悲观锁：**认为线程安全问题一定会发生，因此在操作数据之前先获取锁，确保线程串行执行，

例如Synchronized、Lock都属于悲观锁

**乐观锁：**认为线程安全问题不一定会发生，因此不加锁，只是在更新数据得时候去判断有没有其他线程对数据做了修改，如果没有修改则认为安全，自己才更新数据，如果已经被其他线程修改，说明发生了安全问题，此时可以重试或者异常

乐观锁得关键是判断之前查询到的数据是否被修改过，常见的方式有两种：

版本号法：

|  |  |  |
| --- | --- | --- |
| **Id** | **Stock** | **Version** |
| **7** | **1** | **1** |

**CAS法，**

直接用Stock来做版本

**分布式锁**

因为JVM内部是通过锁监视器来控制Synchronized对象，所以集群得时候会有多个JVM就会有多个锁监视器，导致Synchronized失效

解决方案：共享锁监视器

分布式锁：

多进程可见

高可用

安全性

互斥

高性能

分布式锁的核心是实现多线程之间的互斥，而满足这一点的方式有很多，常见的是三种

|  |  |  |  |
| --- | --- | --- | --- |
|  | MySql | Redis | ZooKeeper |
| 互斥 | 利用mysql本身的互斥锁机制 | 利用setnx的互斥命令 | 利用节点的唯一性和有序性实现互斥 |
| 高可用 | 好 | 好 | 好 |
| 高性能 | 一般 | 好 | 一般 |
| 安全性 | 断开连接，自动释放锁 | 利用锁超时时间，到期释放 | 临时节点，断开连接自动释放 |

基于redis的分布式锁

获取锁：

互斥：确保只能有一个线程获取锁

SET lock thread1 EX 10 NX

非阻塞：尝试一次，成功返回true，失败返回false

释放锁：

手动释放

DEL KEY

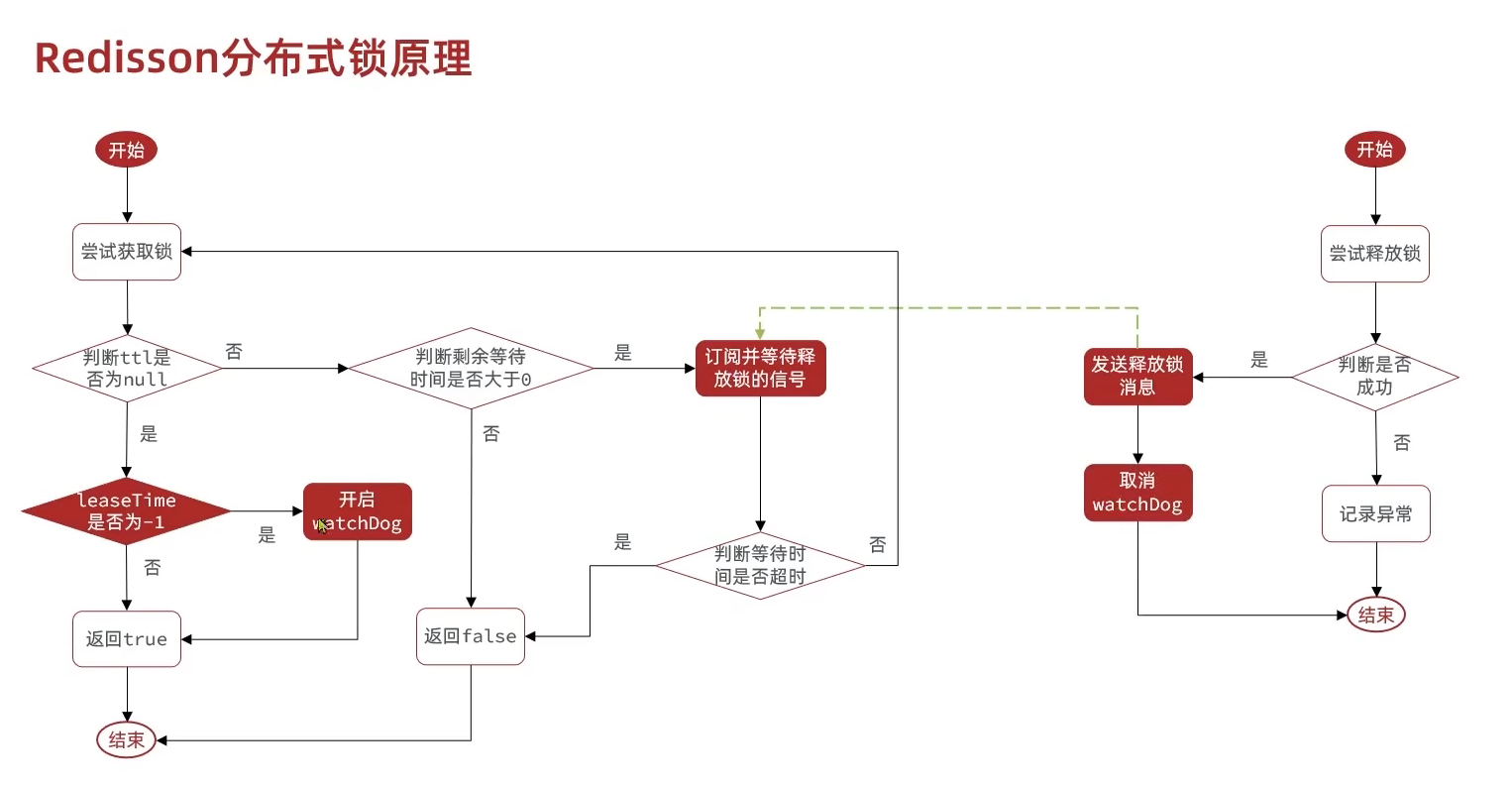
超时释放

释放锁的时候要考虑高并发下的安全问题，用lua脚本来解决redis分布式原子性问题

基于Redis的分布式锁优化

基于setnx实现的分布式锁存在下面的问题

* 1. 不可重入：同一个线程无法多次获取同一把锁
  2. 不可重试：获取锁只尝试一次就返回false，没有重试机制
  3. 超时释放：锁超时释放虽然可以避免死锁，但如果是业务执行耗时较长，也会导致锁释放，存在安全隐患
  4. 主从一致性：如果Redis提供了主机群，主从同步存在延迟，当主宕机时，如果从并同步注重的数据，则会出现锁实现



Redisson分布式锁原理：

可重入：利用hash结构记录线程id和重入次数，

可重试：利用信号量和PubSub功能实现等待，唤醒，获取锁失败的重试机制

超时续约：利用watchdog看门狗，每隔一段时间（releaseTime/3），重置超时时间

Redis主从一致发生的原因

主节点再写入后宕机，从节点没获取锁，导致新的主节点上没有当前线程锁

Redisson解决方案

向redis集群分别获取锁，没有主从之分，当在每一个节点都拿到锁，才成功



提高秒杀业务，提高并发性能

1. 新增秒杀优惠券的同时，将优惠券信息保存到Redis中
2. 基于Lua脚本，判断秒杀库存、一人一单，决定用户是否抢购成功
3. 如果抢购成功，将优惠券id和用户id封装后存入阻塞队列
4. 开启线程任务，不断从阻塞队列中获取信息，实现异步下单功能

秒杀业务的优化思路：

1. 先利用redis完成库存余量、一人一单判断，完成抢单业务
2. 再将下单业务放入阻塞队列，利用独立线程异步下单

基于JDK阻塞队列的异步秒杀存在的问题

1. 内存限制问题
2. 数据安全问题

解决->Redis消息队列实现异步秒杀

消息队列和阻塞队列的区别：

1. 消息队列实在JVM以外独立服务，不受内存限制
2. 消息队列不仅仅是做数据存储，而且确保信息安全，数据不会丢失，消息投递后必须有消费者来确认。确保消息至少被消费一次

Redis提供了三种不同的方式来实现消息队列

List结构：基于List结构模拟消息队列

PubSub：基于点对点的消息模型

Stream：比较完善的消息队列模型（5.0后）

使用List结构模拟队列

队列是入口和出口不在一边，因此我们可以利用：LPUSH结合RPOP、或者RPUSH结合LPOP来实现

※当队列中没有消息是RPOP或LPOP会返回null，并不像JVM阻塞队列一样阻塞并等待消息因此应该使用BRPOP和BLPOP来实现阻塞效果

基于List的消息队列有哪些优缺点

优点：

利用Redis存储，不受限于JVM内存上限

基于Redis的持久化机制，数据安全性有保证

可以满足消费有序性

缺点

无法避免消息丢失

只支持单消费者

基于PubSub的消息队列

PubSub（发布订阅）是2.0版本引入的消息传递模型，消费者可以订阅一个或者多个channel（频道），生产者向对应的channel发送消息后，所有订阅者都能收到相关消息

SUBSCRIBE channel ：订阅一个或者多个频道

PUBLISH channel msg ： 向一个频道发送消息

PSUBSCRIBE pattern：订阅与pattern格式匹配的所有频道

优点：发布订阅模型、支持多生产、多消费

缺点：

不支持数据持久化

无法避免消息丢失

消息堆积有上限，超出时数据丢失

Stream的单消费模式

可以重复消费，消息不会删除

一个消息可以被多个消费者读取

可以阻塞读取

有消息漏读的风险

消费者组：将多尔消费者划分到一个组中，监听同一个队列。具备以下特点

消息分流：队列中的消息会分流给组内的不同消费者，而不是重复消费，从而加快罅隙处理的速度

消息标识：消费者组会维护一个标识，记录最后一个被处理的消息，哪怕消费者宕机重启，还会从标识之后读取消息。确保每一个消息都会被消费。

消息确认：消费者获取消息时，消息处于pending状态，并存入pending-list，当处理完成后需要通过XACK来确认消息，标记消息为已处理，才会从pending-list移除

创建消费者组：

XGROUP CREATE key groupName ID[MKSTREAM]

Key：队列名称

groupName:消费者组名称

ID：起始标识，￥代表队列中最后一个消息，0代表队列中第一个

MKSTREAM：队列不存在时自动创建队列

从消费者组读取消息

XREADGROUP GROUP group consumer [COUNT count] [Block millisecond] [NOACK] Streams key [key …] ID[ID…]

Group：消费组名称

Consumer：消费者名称，如果消费者不存在，会创建一个消费者

Count：本次查询的最大值

Block milliseconds：当没有消息时最长等待时间

NOACK:无需手动ACK,获取消息后自动确认

Streams key：指定队列名称

ID:

“>”:从下一个未消费的消息开始

其他：根据指定id从pending-list中获取已消费但未确认的消息，例如0，是从pending-list中的第一个消息开始

Stream消息队列Xreadgroup命令特点：

消息可回溯

可以多个消费者争抢消息

可以阻塞读取

没有消息漏读风险

有消息确认机制，保证每条消息至少被消费一次

|  |  |  |  |
| --- | --- | --- | --- |
|  | List | PubSub | Stream |
| 消息持久化 | 支持 | 不支持 | 支持 |
| 阻塞读取 | 支持 | 支持 | 支持 |
| 消息堆积处理 | 受限于内存空间，可以利用多个消费者加快处理 | 受限于消费者缓冲区 | 受限于队列长度，可以多个消费者加快处理 |
| 消息确认机制 | 不支持 | 不支持 | 支持 |
| 消息回溯 | 不支持 | 不支持 | 支持 |

解决超卖：乐观锁，悲观锁

解决一人一单限制功能：分布式锁 String类型的分布式锁，redistemplate调用lua脚本，redisson分布式锁看门狗机制hash结构解决锁重入问题，发布订阅解决锁重试

异步秒杀优化：队列

关注推送也叫feed流，直译为投喂，为用户持续得提供沉浸式得体验，通过无限下拉刷新获取新的信息

Feed流产品有两种常见模式：

·TimeLine：不做内容筛选，简单的按照内容发布时间排序，常用于好友或关注，例如朋友圈

优点：信息全面，不会有确实。并且实现也相对简单

缺点：信息噪音较多，用户不一定感兴趣，内容获取效率低

·智能排序：利用智能算法屏蔽掉违规的，用户不感兴趣的内容，推送用户感兴趣信息来吸引用户

优点：投喂用户感兴趣信息，用户黏度高，容易沉迷

缺点：如果算法不精准，可能起到反作用

TimeLine三种实现方案

拉模式：也叫做读扩散



缺点：延迟高

推模式：也叫做写扩散

消息直接写到所有收件箱，然后对收件箱里的消息做排序

推拉结合模式：也叫做读写混合，兼备推拉两种模式

活跃粉丝，采用推，普通粉丝用拉模式，延时无所谓，普通up主则用推，



**Redis数据删除策略**

惰性删除：设置该key过期时间后，我们不去管他，当需要该key时，我们再检查其是否过期，如果过期，我们就删掉他，反之则返回key

优点：对CPU友好，只会在使用该key时才会进行过期检查，对于很多用不到的key不用浪费时间进行过期检查

缺点：对内存不友好，如果一个key已经过期，但是一直没有使用，那么该key就会一直存在内存中，内存永远不会释放

定期删除：每隔一段时间，我们就会对一些key进行检查，删除过期key（随机取出一定数量的key进行检查，并删除过期的key）

定期清除由两种模式：

**·**SLOW模式是定时任务，执行频率默认是10hz，每次不超过25ms，以通过修改配置文件redis.conf的hz选项来调整这个次数

**·**FAST模式执行频率不固定，但两次间隔不低于2ms，每次耗时不超过1ms

优点：可以通过限制删除操作执行的时长和频率来减少删除操作对CPU的影响。另外定期删除也能有效释放过期键占用的内存

缺点：难以确定删除操作执行的时长和频率

Redis淘汰策略：

LFU的访问次数叫做逻辑访问次数，因为并不是每次key被访问都计数，而是通过运算

1. 生成0-1的随机数R
2. 计算1/（旧次数\*因子+1），记录为p，因子默认是10
3. 如果R<p，则计数+1，最大不超过255
4. 访问次数随时间缩减，一分钟-1

LRU：最少最近使用。当前时间减去最后一次使用时间

LFU：最少频率使用，会统计每个key的逻辑访问次数