Redis 使用的 6 种数据结构

在 redis 中有六中数据结构分别应用于 Redis 的各种数据类型中，分别是：

字符串（应用于 String 类型）、

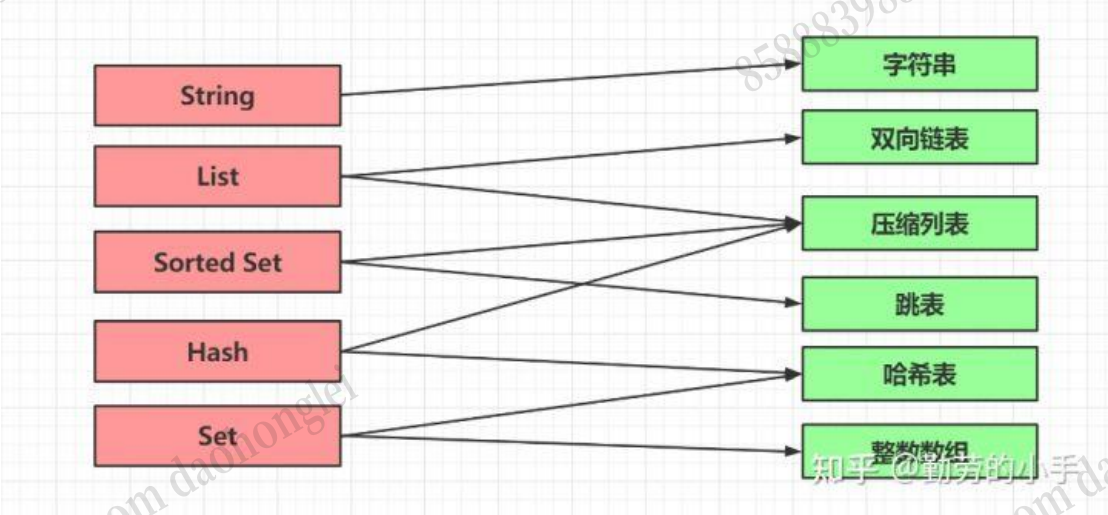
双向链表（应用于 List 类型）、

压缩列表（应用于 List、Sorted Set、Hash 类型）、

跳表（应用于 Sorted Set 类型）、

哈希表（应用于 Set、Hash 类型）、

整数数组（应用于 Set 类型）。



单线程的 redis 为什么这么快

分析:这个问题其实是对 redis 内部机制的一个考察。其实根据博主的面试经验，很多人其实都不知道 redis 是单线程工作模型。所以，这个问题还是应该要复习一下的。

- (一)纯内存操作
- (二)单线程操作，避免了频繁的上下文切换
- (三)采用了非阻塞 I/O 多路复用机制

Redis 过期 key 的删除策略

记录 key 过期时间

- 每个库中都包含了 expires 过期字典（hashtable 结构，键为指针，指向真正 key，值为 long 类型的时间戳，毫秒精度）
- 当设置某个 key 有过期时间时，就会向过期字典中添加此 key 的指针和时间戳

redis 采用了惰性删除 + 定期删除的策略

- 惰性删除
  - 在执行读写数据库的命令时，执行命令前会检查 key 是否过期，如果已过期，则删除 key
- 定期删除
  - redis 有一个定时任务处理器 serverCron，负责周期性任务处理，默认 100 ms 执行一次（hz 参数控制）：

    ① 处理过期 key、② hash 表 rehash、③ 更新统计结果 、④ 持久化、⑤ 清理过期客户端
- 依次遍历库，在规定时间内运行如下操作

  a) 从每个库的 expires 过期字典中随机选择 20 个 key 检查，如果过期则删除

  b) 如果删除达到 5 个，重复 ① 步骤，没有达到，遍历至下一个库

  c) 规定时间没有做完，等待下一轮 serverCron 运行

使用 redis 有什么缺点

分析:大家用 redis 这么久，这个问题是必须要了解的，基本上使用 redis 都会碰到一些问题，常见的面试也会问这几个。

(一)缓存和数据库双写一致性问题

分析:一致性问题是分布式常见问题，还可以再分为最终一致性和强一致性。数据库和缓存双写，就必然会存在不一致的问题。答这个问题，先明白一个前提。就是如果对数据有强一致性要求，不能放缓存。我们所做的一切，只能保证最终一致性。另外，我们所做的方案其实从根本上来说，只能说降低不一致发生的概率，无法完全避免。因此，有强一致性要求的数据，不能放缓存。

首先，采取正确更新策略，先更新数据库，再删缓存。其次，因为可能存在删除缓存失败的问题，提供一个补偿措施即可，例如利用消息队列。

(二)缓存雪崩问题

一般中小型传统软件企业，很难碰到这个问题。如果有大并发的项目，流量有几百万左右。这两个问题一定要深刻考虑。

缓存雪崩，即缓存同一时间大面积的失效，这个时候又来了一波请求，结果请求都怼到数据库上，从而导致数据库连接异常。

解决方案:

① 给缓存的失效时间，加上一个随机值，避免集体失效。

② 使用互斥锁，但是该方案吞吐量明显下降了。

③ 双缓存。我们有两个缓存，缓存 A 和缓存 B。缓存 A 的失效时间为 20 分钟，缓存 B 不设失效时间。自己做缓存预热操作。然后细分以下几个小点

- I 从缓存 A 读数据库，有则直接返回
- II A 没有数据，直接从 B 读数据，直接返回，并且异步启动一个更新线程。
- III 更新线程同时更新缓存 A 和缓存 B。

(三)缓存击穿问题

一般中小型传统软件企业，很难碰到这个问题。如果有大并发的项目，流量有几百万左右。这两个问题一定要深刻考虑。

缓存穿透，即黑客故意去请求缓存中不存在的数据，导致所有的请求都怼到数据库上，从而数据库连接异常。

解决方案:

① 利用互斥锁，缓存失效的时候，先去获得锁，得到锁了，再去请求数据库。没得到锁，则休眠一段时间重试

② 采用异步更新策略，无论 key 是否取到值，都直接返回。value 值中维护一个缓存失效时间，缓存如果过期，异步起一个线程去读数据库，更新缓存。需要做缓存预热(项目启动前，先加载缓存)操作。

③ 提供一个能迅速判断请求是否有效的拦截机制，比如，利用布隆过滤器，内部维护一系列合法有效的 key。迅速判断出，请求所携带的 Key 是否合法有效。如果不合法，则直接返回。

缓存一致性问题

何为旁路缓存（Cache Aside），它是一种使用缓存的策略

查询时的规则

- 先读缓存
- 如果命中，直接返回
- 如果缺失，查 DB 并将结果放入缓存，再返回

增、删、改的规则

- 新增数据，直接存 DB
- 修改、删除数据，先更新 DB，再删缓存

为什么要先操作库，再操作缓存?

- 假设操作库和缓存均能成功，仍会出现数据库与缓存不一致的情况

缓存击穿

缓存击穿是指：某一热点 key 在缓存和数据库中都存在，它过期时，这时由于并发用户特别多，同时读缓存没读到，又同时去数据库去读，压垮数据库

解决方法

1. 热点数据不过期
2. 对【查询缓存没有，查询数据库，结果放入缓存】这三步进行加锁，这时只有一个客户端能获得锁，其它客户端会被阻塞，等锁释放开，缓存已有了数据，其它客户端就不必访问数据库了。但会影响吞吐量（有损方案）

缓存雪崩

情况 1：由于大量 key 设置了相同的过期时间（数据在缓存和数据库都存在），一旦到达过期时间点，这些 key 集体失效，造成访问这些 key 的请求全部进入数据库。

给某个 key 加锁能解决雪崩吗?

解决方法:

1. 错开过期时间:在过期时间上加上随机值(比如 1~5 分钟)
2. 服务降级:暂停非核心数据查询缓存,返回预定义信息(错误页面,空值等)

情况 2:Redis 实例宕机,大量请求进入数据库

解决方法:

1. 事前预防:搭建高可用集群
2. 多级缓存:缺点是实现复杂度高
3. 熔断:通过监控一旦雪崩出现,暂停缓存访问待实例恢复,返回预定义信息(有损方案)
4. 限流:通过监控一旦发现数据库访问量超过阈值,限制访问数据库的请求数(有损方案)

缓存穿透

缓存穿透是指:如果一个 key 在缓存和数据库都不存在,那么访问这个 key 每次都会进入数据库

- 很可能被恶意请求利用
- 缓存雪崩与缓存击穿都是数据库中有,但缓存暂时缺失
- 缓存雪崩与缓存击穿都能自然恢复,但缓存穿透则不能

解决方法:

1. 如果数据库没有,也将此不存在的 key 关联 null 值放入缓存,缺点是这样的 key 没

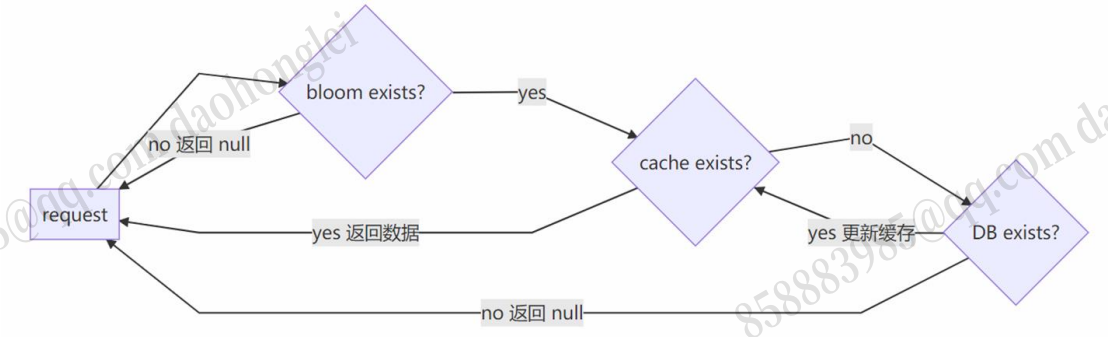
有任何业务作用，白占空间

2. 布隆过滤器

a) 过滤器可以用来判定 key 不存在，发现这些不存在的 key，把它们过滤掉就好

b) 需要将所有的 key 都预先加载至布隆过滤器

c) 布隆过滤器不能删除，因此查询删除的数据一定会发生穿透



布隆过滤器

大家好，我是球球，今天咱们跟大家一起学习一个非常有用的知识：布隆过滤器。他一直在亿级流量的电商系统中的担任着重要角色。“布隆过滤器”这个话题经久流传 ~ 这个牛轰轰的神器是布隆这位大牛在 1970 年发明的，是一个二进制向量数据结构，当时专门用于解决数据查询问题。现在的面试官也特别的钟爱这个神器。

现已 41 岁的布隆过滤器，不仅没有中年危机，反而在江湖中担任着不可替代的角色。

不信的话，我问问你：面试中必问的“布隆过滤器”，工作中最怕的“缓存穿透”，架构中复杂的 Redis，它们之间都是啥关系？

现在不懂没关系，咱们接着往下聊。

Redis 让你网购“剁手”更顺利

你肯定上过京东、淘宝这样的购物网站，不知有没有注意过：在我们日常开发中，其实每

一个页面的 URL 网址是和具体的商品对应的。

比如说，当前我标红的 857，就是我们商品的 sku 编号，你可以理解为是这个商品型号的唯一编码。这里商品编号为 857，那显示的页面自然也是对应的内容。

比如在 618 当天，数亿网友 shopping 剁手时，商城应用同时收到海量请求，要求访问、下单、支付，这时机器如何顶得住呢？这就涉及我们系统的架构了，来看一下。

- 首先作为商城的用户，他发起一个请求，比如这里还是要查看 857 号的商品。

- 这时作为商城应用程序，它会向后台的 Redis 缓存服务器进行查询。

- 如果缓存数据库中没有 857 号的商品数据，我们程序就需要**在后台的数据库服务器中进行查询，并且填充至 Redis 服务器，这是一个正常的操作流程。**

那么在长时间积累后，我们的缓存服务器里面的数据可能就是这个样子的。为了好理解，这里我假设商城有 1000 件商品，编号从 1~1000。

此时此刻作为商城用户，如果查询 857 号商品时，商城应用就不再需要从 MySQL 数据库中进行数据提取，**直接从 Redis 服务器中将数据提取并返回**就可以了。因为 Redis 它是基于内存的，无论是从吞吐量还是处理速度来说，都要比传统的 MySQL 数据库快很多倍。

随着时间不断积累，那么在 Redis 的服务器中应该会存储编号从 1~1000 的所有商品数据缓存。

所以无论是电商还是其他行业，都会在自己的系统架构中增加 Redis 缓存服务器来优化系统的执行速度。

但事无完美，这在当前的设计下会有一个致命问题。

Redis 面临的安全隐患：缓存穿透

大家请注意当前缓存中只有 1~1000 号数据。 假设如果是同行恶意竞争，或者由第三方公司研发了爬虫机器人，在短时间内批量的进行数据查询，而这些查询的编号则是之前数据库中不存在的，比如现在看到的 8888、8889、8890 这些都是不存在的。

此时我们系统就会遇到一个重大的安全隐患：因为商城应用在向后台 Redis 查询时，由于缓存中没有这条数据，它就进而到了数据库服务器中进行查询。

【无效请求超高并发，会导致崩溃】

要知道数据库服务器对于瞬时**超高并发的访问**承载能力并不强。所以在短时间内，由爬虫机器人或者流量攻击机器人发来的这些无效的请求，都会瞬间的灌入到数据库服务器中，对我们的系统的性能造成极大的影响，甚至会产生系统崩溃。

而这种绕过 Redis 服务器，直接进入后台数据库查询的攻击方式，我们就称之为缓存穿透。

对于小规模的缓存穿透是不会对我们系统产生大的影响，但如果是缓存穿透攻击则又是另外一码事了。缓存穿透攻击，是指恶意用户在短时内大量查询不存在的数据，导致大量请求被送达数据库进行查询，当请求数量超过数据库负载上限时，使系统响应出现高延迟甚至瘫痪的攻击行为，就是缓存穿透攻击。

大家不要小看这种攻击方式，在多年前的 618 前夕，我们就遭到了其他平台的恶意缓存穿透攻击，导致商城应用宕机两个小时。造成巨大损失，所以后来我们也增加了对于缓存穿

透攻击的预防。

那么如何预防这种缓存穿透呢?

预防缓存穿透“神器”：布隆过滤器

在架构设计时有一种最常见的设计被称为布隆过滤器，它可以有效减少缓存穿透的情况。

其主旨是采用一个很长的二进制数组，通过一系列的 Hash 函数来确定该数据是否存在。

这么说可能有些晦涩，我们通过一系列的图表演示你就明白了。

布隆过滤器本质上是一个 n 位的二进制数组。你也知道二进制只有 0 和 1 来表示，针对于当前我们的场景。这里我模拟了一个二进制数组，其每一位它的初始值都是 0。

而这个二进制数组会被存储在 Redis 服务器中，那么这个数组该怎么用呢?

1.若干次 Hash 来确定其位置

刚才我们提到作为当前的商城，假设有 1000 个商品编号，从 1~1000。作为布隆过滤器，在初始化的时候，实际上就是对每一个商品编号进行若干次 Hash 来确定它们的位置。

（1）1 号商品计算 比如说针对于当前的“1”编号，我们对其执行了三次 Hash。所谓 Hash 函数就是将数据代入以后确定一个具体的位置。

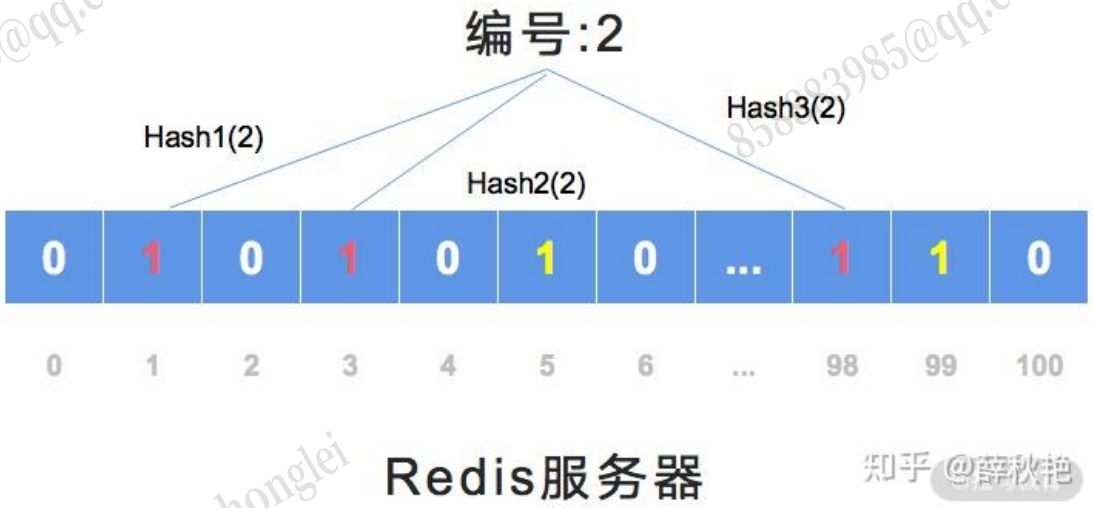
Hash 1 函数：它会定位到二进制数组的第 2 位上，并将其数值从 0 改为 1；

Hash 2 函数：它定位到索引为 5 的位置，并将从 0 改为 1；

Hash 3 函数：定位到索引为 99 的位置上，将其从 0 改为 1。

（2）2 号商品计算

那 1 号商品计算完以后，该轮到 2 号商品。2 号商品经过三次 Hash 以后，分别定位到索引为 1、3 以及 98 号位置上。



原始数据中 1 号位因为刚才已经变成了 1，现在它不变；而 3 号位和 98 号位原始数据从 0 变为 1。

这里又衍生出一个 Hash 新规则：如果在 Hash 后，原始位它是 0 的话，将其从 0 变为 1；如果本身这一位就是 1 的话，则保持不变。

（3）1000 号商品计算

此时 2 号商品也处理完了，我们继续向后 3、4、5、6、7、8 直到编号达到了最后一个 1000，当商品编号 1000 处理完后，他将索引为 3、6、98 设置为 1。

2.布隆过滤器在电商商品中的实践

作为布隆过滤器，它存储在 Redis 服务器中该怎么去使用呢？这就涉及我们日常开发中对于商品编号的比对了。

（1）先看一个已存在的情况 比如，此时某一个用户要查询 858 号商品数据。都知道 858 是存在的，那么按照原始的三个 Hash 分别定位到了 1、5 和 98 号位，当每一个 Hash 位的数值都是 1 时，则代表对应的编号它是存在的。

（2）再看一个不存在的情况 例如这里要查询 8888。8888 这个数值经过三次 Hash 后，

定位到了 3、6 和 100 这三个位置。此时索引为 100 的数值是 0，在多次 Hash 时有任何一位为 0 则代表这个数据是不存在的。

简单总结一下：如果布隆过滤器所有 Hash 的值都是 1 的话，则代表这个数据可能存在。

注意我的表达：它是可能存在；但如果某一位的数值是 0 的话，它是一定不存在的。

布隆过滤器设计之初，它就不是一个精确的判断，因为布隆过滤器存在误判的情况。

（3）最后看一个误判情况 来看一下当前的演示：比如现在我要查询 8889 的情况，经过三次 Hash 正好每一位上都是 1。尽管在数据库中，8889 这个商品是不存在的；但在布隆过滤器中，它会被判定为存在。这就是在布隆过滤器中会出现的小概率的误判情况。

3.如何减少布隆过滤器的误判?

关于减少误判的产生，方法有两个：

- 第一个是增加二进制位数。在原始情况下我们设置索引位到达了 100，但是如果我们把它放大 1 万倍，到达了 100 万，是不是 Hash 以后的数据会变得更分散，出现重复的情况就会更小，这是第一种方式。

- 第二个是增加 Hash 的次数。其实每一次 Hash 处理都是在增加数据的特征，特征越多，出现误判的概率就越小。

现在我们是做了三次 Hash，那么如果你做十次，是不是它出现误判的概率就会小非常多？但是在这个过程中，代价便是 CPU 需要进行更多运算，这会让布隆过滤器的性能有所降低。

讲到这里，想必你对布隆过滤器应该有所了解了。但是在我们开发过程中，我们如何去使用布隆过滤器？来咱们看一下。

开发中，如何使用布隆过滤器?

- 1.布隆过滤器在 Java 中的应用 其实作为 Java 积累了这么多年，像布隆过滤器这种经典的算法，早就为我们进行了封装和集成。在 Java 中提供了一个 Redisson 的组件，它内置了布隆过滤器，可以让程序员非常简单直接地去设置布隆过滤器。

上面代码是 Redisson 的使用办法,在前几行代码,用来设置 Redis 服务器的服务地址及端口号。

而后面关键的地方在这里，我们实例化一个布隆过滤器对象，后面的参数指代 Redis 使用哪个 key 来保存布隆过滤器数据?

下面这句话非常关键，作为当前的布隆过滤器，这里需要调用 tryInit 方法，它有两个参数：

第一个参数是代表初始化的布隆过滤器长度，长度越大，出现误判的可能性就越低。

而第二个 0.01 则代表误判率最大允许为 1%，在我们以前的项目中通常也是设置为 1%。

如果把这个数值设置太小，虽然会降低误判率，但会产生更多次的 Hash 操作，会降低系统的性能（正是刚刚讲过的），因此 1% 也是我所建议的数值。

当把布隆过滤器初始化以后，我们便可以通过 add 方法，往里边去添加数据。所谓添加数据，就是将数据进行多次 Hash，将对应位从 0 变为 1 的过程。例如，现在我们把编号 1 增加进去，之后可以通过布隆过滤器的 contains 方法来判断当前这个数据是否存在。

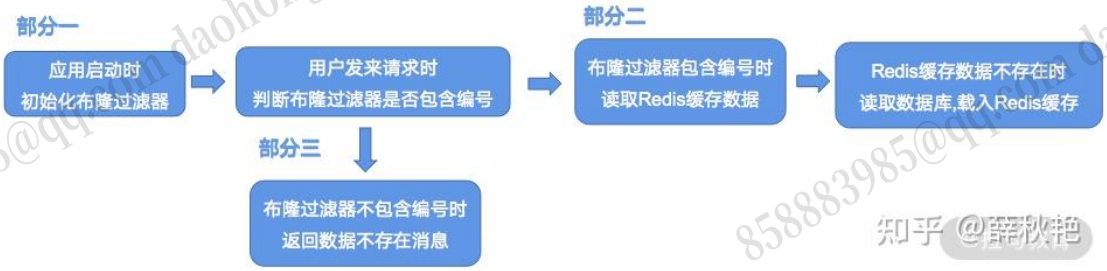
我们输入 1，它输出 true；而输入了不存在的 8888，则输出 false。 请注意：这两个结果的含义是不同的。

如果输出 false，则代表这个数据它是肯定不存在的；

但是如果输出 true 的时候，它有 1% 的概率可能不存在，因为布隆过滤器它存在误判的情况。

以上便是布隆过滤器在 Java 中的应用，但是布隆过滤器如果要运用在项目中又该变成什么样子？它的处理流程是什么？

- 2.布隆过滤器在项目中的应用 咱们看一下布隆过滤器在项目中的使用流程，其实就归结成以下三部分：



第一个部分是在应用启动时，我们去初始化布隆过滤器。例如将 1000 个、1 万个、10 万个商品进行初始化，完成从 0 到 1 的转化工作。

之后便是当用户发来请求时，会附加商品编号，如果布隆过滤器判断编号存在，则直接去读取存储在 Redis 缓存中的数据；如果此时 Redis 缓存没有存在对应的商品数据，则直接去读取数据库，并将读取到的信息重新载入到 Redis 缓存中。这样下一次用户在查询相同编号数据时，就可以直接读取缓存了。

另外一种情况是，如果布隆过滤器判断没有包含编号，则直接返回数据不存在的消息提示，这样便可以在 Redis 层面将请求进行拦截。

你可能会有疑惑，既然布隆过滤器存在误判率，那出现了误判该怎么办呢？

其实在大多数情况下，我们出现误判也不会对系统产生额外的影响。因为像刚才我们设置 1% 的误判率，1 万次请求才可能会出现 100 次误判的情况。我们已经将 99% 的无效请求进行了拦截，而这些漏网之鱼也不会对我们系统产生任何实质影响。

延伸问题：初始化后，对应商品被删怎么办？ 最后还有一个延伸的小问题：假如布隆过滤

器初始化后，对应商品被删除了，该怎么办呢？这是一个布隆过滤器的小难点。

因为布隆过滤器某一位的二进制数据，可能被多个编号的 Hash 位进行引用。比如说，布隆过滤器中 2 号位是 1，但是它可能被 3、5、100、1000 这 4 个商品编号同时引用。这里是不允许直接对布隆过滤器某一位进行删除的，否则数据就乱了，怎么办呢？

这里业内有两种常见的解决方案：

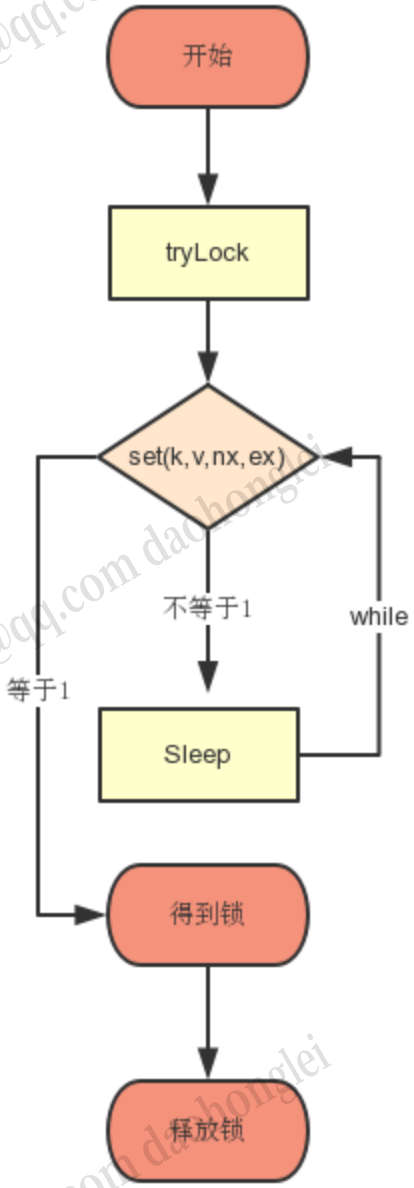
定时异步重建布隆过滤器。比如说我们每过 4 个小时在额外的一台服务器上，异步去执行一个任务调度，来重新生成布隆过滤器，替换掉已有的布隆过滤器。

计数布隆过滤器。在标准的布隆过滤器下，是无法得知当前某一位它是被哪些具体数据进行了引用，但是计数布隆过滤器它是在这一位上额外的附加的计数信息，表达出该位被几个数据进行了引用。（如果你对计数布隆过滤器有兴趣的话，可以再去翻阅一下相关资料）

Redis + Lua 实现分布式锁(SpringBoot 版)

设计思路

既然是实现分布式锁，那肯定得保证多个连接集中请求一个资源的排他性,而 redis 的单线程特性则很好的满足了这一需求。redis 提供的 set 方法则是满足这一需求的关键，下图是实现 redis 分布式锁的简单流程，先有个初步的料及。



场景分析

下面是 set 命令的相关用法：

```
SET key value [EX seconds] [PX milliseconds] [NX|XX]
```

可选参数:

- EX second ：设置键的过期时间为 second 秒。 SET key value EX second 效果等同于 SETEX key second value 。
- PX millisecond ：设置键的过期时间为 millisecond 毫秒。 SET key value PX millisecond 效果等同于 PSETEX key millisecond value 。
- NX ：只在键不存在时，才对键进行设置操作。 SET key value NX 效果等同于 SETNX key value 。
- XX ：只在键已经存在时，才对键进行设置操作。

看似简单的实现，实则有很多隐藏的坑,下面我将以几个案例作为分析.

Case1 :

熟悉 redis 命令的同学可能注意到 setnx 这个命令，然后再配合 expire 命令是否可以实现一样的效果了？

```
SETNX user_id 10086
// 此处服务挂掉了，那么 user_id 将永生
expire user_id
```

注意，这是 2 条命令也就意味着这是非原子性操作，当执行到第一条命令 SETEX user_id 10086，再准备给这个 key 设置失效时间时服务突然挂掉了，那么这个 key 将会永生，其他请求将永远无法获取到这把锁。

所以，我们需要用 SET 保证其原子性. SET user_id 10086 EX 30 NX，即设置 key 为 user_id,value 为 10086，并且设置失效时间为 30s，如果该 key 存在则放弃更改。

Case2 :

在我们设置 key 值的时候一般会尽量保证他的唯一性，比如订单 ID，库存 ID 等。而根据 SET 命令的返回结果来判断是否有其他请求强占，貌似 value 值的设置可有可无，事实真的如此吗？

伪代码：

```
SET user_id 10086 EX 30 NX
// 处理业务中


//业务处理完毕
DEL user_id
```

流程图分析:

通过上图我们将请求的过程肢解，即分为以下几步：

1. 三个请求 A,B,C 同时竞争锁，被请求 A 抢先获得，其他请求只能不断尝试获取锁 (tryLock)
2. 请求 A 由于业务比较复杂处理时间已经超时，所以请求 B 能够获取到锁
3. 请求 A 终于完成了自己的业务，这个时候执行了 DEL user_id，但是他自己的锁已经失效了，删除的是请求 B 锁。而请求 B 的业务此时并未处理完，所以此处就出现了问题！

改进：

为了避免误删除别人的锁，所以我们需要在删除锁的时候需要判断一下这个锁是否是自己的。这个时候我们设置的 value 就生效了,可以通过 value 来判断这把锁是否属于自己。 这个 value 值设置比较随意，只要能做区分就可以了。

伪代码:

```
SET user_id 10086 EX 30 NX
// 处理业务中



//业务处理完毕
if( (GET user_id) == "XXX" ){
  DEL user_id
}
```

Case3 :

好吧，终于解决了这一系列坑本以为就要完工 。正在得意回味自己改进的代码时总觉得哪里有点怪怪的，猛地发现这个 GET 取值判断和 DEL 删除并非原子操作。那么接着上面的分析，会出现什么问题呢？

```
if( (GET user_id) == "XXX" ){ //获取到自己锁后，进行取值判断且判断为真。此时，这把锁恰好失效。
  DEL user_id
}
```

当程序判通过该锁的值判断发现这把锁是自己加上的，准备 DEL。此时该锁恰好失效，而另外一个请求恰好获得 key 值为 user_id 的锁。

此时程序执行了了 DEL user_id，删除了别人加的锁，尴尬！

所以这段代码并不完美，为了保证查询和删除的原子性操作，需要引入 lua 脚本支持。

改进：

```
String luaScript = "if redis.call('get', KEYS[1]) == ARGV[1] then return redis.call('del', KEYS[1])
else return 0 end";
jedis.eval(luaScript , Collections.singletonList(key), Collections.singletonList(threadId));
```

补充

回到 Case2 中，请求 A 中的业务处理时长超过了锁的失效时间。对于此类问题，看到很多网上大佬给出的答案是起一个守护线程进行监听 key 的失效时间，然后在快要失效的时候为期续命。

其实 redis 对于分布式锁，Redisson 有着更好的实现方式。

Lua 实现分布式锁

加锁:

```
--[[
思路：
    1.用 2 个局部变量接受参数

    2.由于 redis 内置 lua 解析器，执行加锁命令

    3.如果加锁成功，则设置超时时间
```

```
    4.返回加锁命令的执行结果
]]
local key = KEYS[1]
local value = KEYS[2]
local rs1 = redis.call('SETNX',key,value)
if rs1 == true
then
    redis.call('SETEX', key,3600, value)
end
return rs1
```

解锁：

```
--[[
思路：
    1.接受 redis 传来的参数
    2.判断是否是自己的锁，是则删掉
    3.返回结果值
]]
local key = KEYS[1]
local value = KEYS[2]


if redis.call('get',key) == value
then
    return redis.call('del',key)
else
    return false
end
```

秒杀：

```
if tonumber(redis.call('get',KEYS[1]))>0
then
    redis.call('decr',KEYS[1]])
    redis.call('lpush',KESY[2],ARGV[1])
    return 1
else
    return 0
end
```

redis 有一亿个 key，使用 keys 命令是否会影响线上服务

- keys 命令时间复杂度是 O(n)，n 即总的 key 数量，n 如果很大，性能非常低
- redis 执行命令是单线程执行，一个命令执行太慢会阻塞其它命令，阻塞时间长甚至会让 redis 发生故障切换
- 可以使用 scan 命令替换 keys 命令
    a) 虽然 scan 命令的时间复杂度仍是 O(n)，但它是通过游标分步执行，不会导致长时间阻塞
    b) 可以用 count 参数提示返回 key 的个数
    c) 返回值代表下次的起点（桶下标）
    d) scan 能保证在 rehash 也正常工作
    e) 弱状态，客户端仅需维护游标
    f) 缺点是可能会重复遍历 key（缩容时）、应用应自己处理重复 key

Redis 的持久化机制是什么？各自的优缺点？

redis 提供两种持久化机制 RDB 和 AOF 机制。

1）RDB 持久化方式：

是指用数据集快照的方式记录 redis 数据库的所有键值对。

优点：

1.只有一个文件 dump.rdb，方便持久化。

2.容灾性好，一个文件可以保存到安全的磁盘。

3.性能最大化，fork 子进程来完成写操作，让主进程继续处理命令，所以是 IO 最大化。

4.相对于数据集大时，比 AOF 的启动效率更高。

缺点：

1.数据安全性低。

2）AOF 持久化方式：

是指所有的命令行记录以 redis 命令请求协议的格式保存为 aof 文件。

优点：

1.数据安全，aof 持久化可以配置 appendfsync 属性，有 always，每进行一次命令操作就记录到 aof 文件中一次。

2.通过 append 模式写文件，即使中途服务器宕机，可以通过 redis-check-aof 工具解决数据一致性问题。

3.AOF 机制的 rewrite 模式。

缺点：

1.文件会比 RDB 形式的文件大。

2.数据集大的时候，比 rdb 启动效率低。

Redis 持久化 – AOF

- AOF - 将每条写命令追加至 aof 文件，当重启时会执行 aof 文件中每条命令来重建内存数据
- AOF 日志是写后日志，即先执行命令，再记录日志
  - Redis 为了性能，向 AOF 记录日志时没有对命令进行语法检查，如果要先记录日志，那么日志里就会记录语法错误的命令
- 记录 AOF 日志时，有三种同步策略

g) Always，同步写，日志写入磁盘再返回，可以做到基本不丢数据，性能不高

- 为什么说基本不丢呢，因为 aof 是在 serverCron 事件循环中执行 aof 写入的，并且这次写入的上一次循环暂存在 aof 缓冲中的数据，因此最多还是可能丢失一个循环的数据

h) Everysec，每秒写，日志写入 AOF 文件的内存缓冲区，每隔一秒将内存缓冲区数据刷入磁盘，最多丢一秒的数据

i) No，操作系统写，日志写入 AOF 文件的内存缓冲区，由操作系统决定何时将数据刷入磁盘

Redis 持久化 – AOF 重写

AOF 文件太大引起的问题

1. 文件大小受操作系统限制

2. 文件太大，写入效率变低

3. 文件太大，恢复时非常慢

AOF 重写

1. 重写就是对同一个 key 的多次操作进行瘦身

   a) 例如一个 key 我改了 100 遍，aof 里记录了 100 条修改日志，但实际上只有最后一次有效

   b) 重写无需操作现有 aof 日志，只需要根据当前内存数据的状态，生成相应的命令，记入一个新的日志文件即可

   c) 重写过程是由另一个后台子进程完成的，不会阻塞主进程

2. 当 AOF 重写发生时

   a) 创建子进程时会根据主进程生成内存快照，只需要对子进程的内存进行遍历，把每个 key 对应的命令写入新的日志文件（即重写日志）

   b) 此时如果有新的命令执行，修改的是主进程内存，不会影响子进程内存，并且新命令会记录到 `重写缓冲区`

   c) 等子进程所有的 key 处理完毕，再将 `重写缓冲区` 记录的增量指令写入重写日志

   d) 在此期间旧的 AOF 日志仍然在工作，待到重写完毕，用重写日志替换掉旧的 AOF 日志

Redis 持久化 – RDB

RDB - 是把整个内存数据以二进制方式写入磁盘

- 对应数据文件为 `dump.rdb`
- 好处是恢复速度快

相关命令有两个

1. save - 在主进程执行，会阻塞其它命令
2. bgsave - 创建子进程执行，避免阻塞，是默认方式

- 子进程不会阻塞主进程，但创建子进程的期间，仍会阻塞，内存越大，阻塞时间越长
- bgsave 也是利用了快照机制，执行 RDB 持久化期间如果有新数据写入，新的数据修改发生在主进程，子进程向 RDB 文件中写入还是旧的数据，这样新的修改不会影响到 RDB 操作
- 但这些新数据不会补充至 RDB 文件

缺点： save 参数可以控制 rdb 的执行周期，但这个周期不好把握

- 频繁执行影响性能
- 偶尔执行，如果宕机又容易丢失较多数据

Redis 持久化 – 混合持久化

从 4.0 开始，Redis 支持混合持久化，即使用 RDB 作为全量备份，两次 RDB 之间使用 AOF 作为增量备份

1. 配置项 aof-use-rdb-preamble 用来控制是否启用混合持久化，默认值 no
2. 持久化时将数据都存入 AOF 日志，日志前半部分为二进制的 RDB 格式，后半部分是 AOF 命令日志
3. 下一次 RDB 时，会覆盖之前的日志文件

优缺点

1. 结合了 RDB 与 AOF 的优点，恢复速度快，增量用 AOF 表示，数据更完整（取决于同步策略）、也无需 AOF 重写

2. 与旧版本的 redis 文件格式不兼容

缓存原子性

单条命令是原子性，这是由 redis 单线程保障的，多条命令能否用 multi + exec 来保证其原子性呢?

对 Redis 中 multi + exec 的认识

1. multi + exec 并不支持回滚，例如

   set a 1000，set b 1000 set c a

   multi，decr a，incr b，incr c，exec

2. multi + exec 中的读操作没有意义

   3. 既然 multi + exec 中读没有意义，就无法保证读+写的原子性，例如

   set a 1000，set b 1000

   get a，get b（分别为 1000），现在转账 500

   multi，set a 500，set b 1500，exec

   但如果在 get 与 multi 之间其它客户端修改了 a 或 b，会造成丢失更新

用乐观锁保证原子性

watch 命令，用来盯住 key（一到多个），如果这些 key 在事务期间:

- 没有被别的客户端修改，则 exec 才会成功
- 被别的客户端改了，则 exec 返回 nil

例如

   set a 1000，set b 1000

watch a b

multi

set a 500，set b 1500

exec

用 Watch 机制实现秒杀

思路：

利用 redis 的 watch 功能，监控这个 redisKey 的状态值

获取 redisKey 的值

创建 redis 事务

给这个 key 的值+1

然后去执行这个事务，如果 key 的值被修改过则回滚，key 不加 1

实现代码：

```
public static void main(String[] arg) {
   String redisKey = "lock";
   ExecutorService executorService = Executors.newFixedThreadPool(20);//20 个线
程
   try {//初始化
      Jedis jedis = new Jedis("127.0.0.1", 6379);
      // 初始值
      jedis.set(redisKey, "0");
      jedis.close();
```

```
    } catch (Exception e) {
        e.printStackTrace();
    }

    for (int i = 0; i < 1000; i++) {//尝试 1000 次
        executorService.execute(() -> {
            Jedis jedis1 = new Jedis("127.0.0.1", 6379);
            try {
                jedis1.watch(redisKey);
                String redisValue = jedis1.get(redisKey);
                int valInteger = Integer.valueOf(redisValue);
                String userInfo = UUID.randomUUID().toString();
                // 没有秒完
                if (valInteger < 20) {//redisKey
                    Transaction tx = jedis1.multi();//开启事务
                    tx.incr(redisKey);//自增
                    List list = tx.exec();//提交事务，如果返回 nil 则说明执行失败，因为我
watch 了的，只要执行失败，则
                    // 进来发现东西还有，秒杀成功
                    if (list != null && list.size() > 0) {
                        System.out.println("用户：" + userInfo + "，秒杀成功！当前成功人数:
" + (valInteger + 1));
```

```
                }else {//执行结果不是 OK，说明被修改了，被别人抢了
                    System.out.println("用户：" + userInfo + "，秒杀失败");
                }
            }else {//东西秒完了
                System.out.println("已经有 20 人秒杀成功，秒杀结束");
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {//关闭 redis
            jedis1.close();
        }
      });
   }
   executorService.shutdown();//关闭线程池
}
```

用 lua 脚本保证原子性

redis 支持 lua 脚本，能保证 lua 脚本执行的原子性，可以取代 multi + exec

例如：eval "local a = tonumber(redis.call('GET',KEYS[1]));local b = tonumber(redis.call('GET',KEYS[2]));local c = tonumber(ARGV[1]); if(a >= c) then redis.call('SET', KEYS[1], a-c); redis.call('SET', KEYS[2], b+c); return 1;else return 0;

end" 2 a b 500

```
local a = tonumber(redis.call('GET',KEYS[1]));
local b = tonumber(redis.call('GET',KEYS[2]));
local c = tonumber(ARGV[1]);
if(a >= c) then
    redis.call('SET', KEYS[1], a-c);
    redis.call('SET', KEYS[2], b+c);
    return 1;
else
    return 0;
end
```

LRU Cache

Least Recently Used，将最近最少使用的 key 从缓存中淘汰掉

- 时间上，新的留下，老的淘汰
- 如果访问了某个 key，则它就变成最新的

实现策略：

1. 链表法，最近访问的 key 移动到链表头，不常访问的自然靠近链表尾，如果超过容量、个数限制，移除尾部的
2. 随机取样法，链表法占用内存较多，redis 使用的是随机取样法，每次只抽 5 个 key，每个 key 记录了它们的最近访问时间，在这 5 个里挑出最老的移除

redis 锁实现

```
import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.data.redis.core.StringRedisTemplate;

import org.springframework.data.redis.core.script.DefaultRedisScript;

import org.springframework.data.redis.core.script.RedisScript;

import org.springframework.stereotype.Component;


import java.time.LocalDateTime;

import java.util.Arrays;

import java.util.Collections;

import java.util.List;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.TimeUnit;


/**

 * @Title: RedisComponent

 * @Author: 181514

 * @Date 2023/5/11 13:34

 */
```

```
@Component
public class RedisComponent {
    //redis 锁过去时间
    private int BASE_EXPIRE_TIMES = 30;
    //最大从事次数
    private int MAX_RETRY_TIMES = 5;
    // 最大重试间隔时间(分钟)
    private int RETRY_INTERVAL_TIMES = 5;

    private ExecutorService executorService = Executors.newFixedThreadPool(2);

    @Autowired
    private StringRedisTemplate stringRedisTemplate;

    public boolean lock(String key, String value, int... expireTimes) {
        if (expireTimes == null || expireTimes.length == 0 || expireTimes[0] <= 0) {
            expireTimes = new int[]{BASE_EXPIRE_TIMES};
        }
        long beginTimes = System.currentTimeMillis();
        while (((System.currentTimeMillis() - beginTimes) / 1000) < expireTimes[0]) {
            if (stringRedisTemplate.opsForValue().setIfAbsent(key, value,
expireTimes[0], TimeUnit.SECONDS)) {
```

```
            executorService.submit(() -> {
                while (true) {
                    if (value.equals(stringRedisTemplate.opsForValue().get(key))) {
                        Long expire = stringRedisTemplate.getExpire(key);
                        if (expire <= 5) {
                            stringRedisTemplate.expire(key, 15, TimeUnit.SECONDS);
                        }
                    } else {
                        return;
                    }
                }
            });
            return true;
        }
    }
    return false;
}

public void unlock(String key, String value) {
    String luaScript = " if redis.call('get',KEYS[1])==ARGV[1] then return
redis.call('del',KEYS[1]) else return 0 end";
    RedisScript<Boolean> script = new DefaultRedisScript<>(luaScript,
```

```
Boolean.class);
        List<String> keys = Collections.singletonList(key);
        stringRedisTemplate.execute(script, keys, value);

        /*if (value.equals(stringRedisTemplate.opsForValue().get(key))) {
            stringRedisTemplate.delete(key);
        }*/
    }

    public boolean reentrantLock(String key, String value) {
        long beginTimes = System.currentTimeMillis();
        while (((System.currentTimeMillis() - beginTimes) / 1000) <
BASE_EXPIRE_TIMES) {
            String luaScript = "if redis.call('exists', KEYS[1])==1 then " +
                    "    if redis.call('hexists', KEYS[1], KEYS[2])==1 then " +
                    "        redis.call('hincrby', KEYS[1] , KEYS[2] ,1) ;" +
                    "        return 1 " +
                    "    else " +
                    "        return 0 " +
                    "    end " +
                    "else " +
                    "    redis.call('hset', KEYS[1] , KEYS[2], 1) " +
```

```
            "   return redis.call('expire', KEYS[1], ARGV[1])" +
            "end";
        RedisScript<Boolean> script = new DefaultRedisScript<>(luaScript,
Boolean.class);
        List<String> keys = Arrays.asList(key, value);
        boolean result = stringRedisTemplate.execute(script, keys,
String.valueOf(BASE_EXPIRE_TIMES));
        if (result){
            String string = (String) stringRedisTemplate.opsForHash().get(key, value);
            if ("1".equals(string)){
                executorService.submit(() -> {
                    while (true) {
                        if (stringRedisTemplate.opsForHash().hasKey(key,value)) {
                            Long expire = stringRedisTemplate.getExpire(key);
                            if (expire <= 5) {
                                stringRedisTemplate.expire(key, 15, TimeUnit.SECONDS);
                            }
                        } else {
                            return;
                        }
                    }
                });
```

```
            }
            return true;
        }
    }
    return false;
}

public boolean reentrantUnLock(String key, String value) {
    String luaScript = "if redis.call('exists', KEYS[1])==1 then " +
            "    if redis.call('hexists', KEYS[1], KEYS[2])==1 then " +
            "        if tonumber(redis.call('hincrby', KEYS[1] , KEYS[2] ,-1))<=0 then " +
            "          redis.call('hdel', KEYS[1] , KEYS[2]) " +
            "          redis.call('del',  KEYS[1]) " +
            "          return 0 " +
            "        else " +
            "           return 1 " +
            "        end" +
            "    else " +
            "        return 0 " +
            "    end " +
            "else " +
            "    return 0 " +
```

```
            "end";
        RedisScript<Boolean> script = new DefaultRedisScript<>(luaScript,
Boolean.class);
        List<String> keys = Arrays.asList(key, value);
        return stringRedisTemplate.execute(script, keys);
    }



    public boolean isExceedMaxRetryTimes(String name) {
        int failedTimes = 0;
        for (int i = 0; i < RETRY_INTERVAL_TIMES; i++) {
            Object object = stringRedisTemplate.opsForHash().get("login" + i, name);
            if (object == null) {
                continue;
            }
            failedTimes = failedTimes + Integer.parseInt((String) object);
            if (failedTimes >= MAX_RETRY_TIMES) {
                return true;
            }
        }
        return false;
    }
```

```
    public void loginSuccess(String name) {
        for (int i = 0; i < RETRY_INTERVAL_TIMES; i++) {
            stringRedisTemplate.opsForHash().delete("login" + i, name);
        }
    }

    public void loginFailed(String name) {
        LocalDateTime localDateTime = LocalDateTime.now();
        int minute = localDateTime.getMinute();
        String key = "login" + (minute % RETRY_INTERVAL_TIMES);
        if (stringRedisTemplate.hasKey(key)) {
            if (stringRedisTemplate.opsForHash().hasKey(key, name)) {
                int times = Integer.parseInt((String)
stringRedisTemplate.opsForHash().get(key, name));
                stringRedisTemplate.opsForHash().put(key, name,
String.valueOf(++times));
            } else {
                stringRedisTemplate.opsForHash().put(key, name, String.valueOf(1));
            }
        } else {
            // lua 脚本实现
```

```
        String luaScript = "if redis.call('hset', KEYS[1] , KEYS[2] , 1) == 1 then return
redis.call('expire', KEYS[1], ARGV[1]) end";
        RedisScript<Boolean> script = new DefaultRedisScript<>(luaScript,
Boolean.class);
        List<String> keys = Arrays.asList(key, name);
        Boolean result = stringRedisTemplate.execute(script, keys,
String.valueOf(RETRY_INTERVAL_TIMES * 60));
        System.out.println(script.getScriptAsString());
        System.out.println(script.getResultType());
        System.out.println(script.getSha1());
        if (!result) {
            stringRedisTemplate.opsForHash().delete(key, name);
        }
        // 事务实现
        /*stringRedisTemplate.execute(new SessionCallback<Boolean>() {
            @Override
            public Boolean execute(RedisOperations operations) throws
DataAccessException {
                operations.multi();
                stringRedisTemplate.opsForHash().put(key, name, String.valueOf(1));
                stringRedisTemplate.expire(key, RETRY_INTERVAL_TIMES,
TimeUnit.MINUTES);
```

```
                List<Boolean> exec = operations.exec();
                Boolean transactionResult = exec.stream().reduce(true, (preResult,
currentVal) -> preResult && currentVal);
                return transactionResult;
            }
        });*/
    }
}
```

```
import com.demo.config.RedisComponent;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.Random;

/**
 * @Title: TestController
 * @Author: 181514
 * @Date 2023/5/11 13:53
 */
```

```
@RestController
public class TestController {

    @Autowired
    private RedisComponent redisComponent;

    Random random = new Random();

    @GetMapping("lock")
    public String test() {
        if (redisComponent.lock("lockTest", Thread.currentThread().getName())) {
            try {
                System.out.println(Thread.currentThread().getName() + " lock success");
                int i = random.nextInt(50) * 1000;
                System.out.println(Thread.currentThread().getName() + " do work  " + i);
                Thread.sleep(i);
                System.out.println(Thread.currentThread().getName() + " lock release");
            } finally {
                redisComponent.unlock("lockTest", Thread.currentThread().getName());
                return "lock success";
            }
```

```
        } else {
            System.out.println(Thread.currentThread().getName() + " lock failed");
            return "lock failed";
        }
    }

    @GetMapping("reentrantLock")
    public String reentrantLock() {
        System.out.println(Thread.currentThread().getName());
        if (redisComponent.reentrantLock("reentrantLock",
Thread.currentThread().getName())) {
            try {
                if (redisComponent.reentrantLock("reentrantLock",
Thread.currentThread().getName())) {
                    System.out.println(Thread.currentThread().getName() + "
reentrantLock success");
                    redisComponent.reentrantUnLock("reentrantLock",
Thread.currentThread().getName());
                } else {
                    System.out.println(Thread.currentThread().getName() + "
reentrantLock failed");
                }
```

```
            System.out.println(Thread.currentThread().getName() + " lock success");
            int i = random.nextInt(50) * 1000;
            System.out.println(Thread.currentThread().getName() + " do work  " + i);
            Thread.sleep(i);
            System.out.println(Thread.currentThread().getName() + " lock release");
        } finally {
            redisComponent.reentrantUnLock("reentrantLock",
Thread.currentThread().getName());
            return "lock success";
        }
    } else {
        System.out.println(Thread.currentThread().getName() + " lock failed");
        return "lock failed";
    }
  }


  @GetMapping("login")
  public String login(String name, String pwd) {
    if (redisComponent.isExceedMaxRetryTimes(name)) {
        return "超过最大错误次数";
    }
```

```
        if ("张三".equals(name) && "123".equals(pwd)) {
            redisComponent.loginSuccess(name);
            return "login success";
        } else {
            redisComponent.loginFailed(name);
            return "login failed";
        }
    }
}
```

redis 限流

```
import com.ldh.utils.exception.RedisLimitException;
import lombok.extern.slf4j.Slf4j;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;
import org.aspectj.lang.reflect.MethodSignature;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.core.io.ClassPathResource;
import org.springframework.data.redis.core.StringRedisTemplate;
import org.springframework.data.redis.core.script.DefaultRedisScript;
```

```
import org.springframework.scripting.support.ResourceScriptSource;
import org.springframework.stereotype.Component;


import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import java.lang.annotation.*;
import java.lang.reflect.Method;
import java.util.*;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;


@Slf4j
@Aspect
@Component
public class RedisLimitAop {


    @Autowired
    private StringRedisTemplate stringRedisTemplate;


    private DefaultRedisScript<Boolean> redisScript;
    private DefaultRedisScript<Boolean> redisConsumptionScript;
```

```
    private ExecutorService executorService = Executors.newFixedThreadPool(4);



    @Pointcut("@annotation(com.example.demo3.comp.RedisLimitAop.Limit)")
    private void check() {
    }



    @PostConstruct
    public void init() {
        redisScript = new DefaultRedisScript<>();
        redisScript.setResultType(Boolean.class);
        redisScript.setScriptSource(new ResourceScriptSource(new
ClassPathResource("lua/redisLimit.lua")));

        redisConsumptionScript = new DefaultRedisScript<>();
        redisConsumptionScript.setResultType(Boolean.class);
        redisConsumptionScript.setScriptSource(new ResourceScriptSource(new
ClassPathResource("lua/redisLimitConsumption.lua")));

        for (int i = 0; i <= 3; i++) {
            int finalI = i;
```

```
        executorService.submit(() -> {
            while (true) {
                try {
                    Map<Object, Object> funcLimit =
stringRedisTemplate.opsForHash().entries("funcLimit" + finalI);
                    Set<Map.Entry<Object, Object>> entrySet = funcLimit.entrySet();
                    for (Map.Entry<Object, Object> entry : entrySet) {
                        String hKey = entry.getKey().toString();
                        int index = hKey.lastIndexOf(":");
                        Long addLimit =
Long.valueOf(hKey.substring(hKey.lastIndexOf(":", index - 1) + 1, index));
                        Integer permit = Integer.parseInt(entry.getValue().toString());
                        Long increment = Long.valueOf(hKey.substring(index + 1));
                        if (increment > 0 && addLimit >= permit) {
                            stringRedisTemplate.opsForHash().increment("funcLimit" +
finalI, entry.getKey(), increment);
                        }
                    }
                    Thread.sleep(1000);
                } catch (Exception e) {
                    log.error(Thread.currentThread().getName() + " 添加令牌出错 ", e);
                }
```

```
            }
        });
    }
}

@PreDestroy
public void destroy() {
    for (int i = 0; i <= 3; i++) {
        stringRedisTemplate.delete("funcLimit" + i);
        stringRedisTemplate.delete("funcLimitConsumption" + i);
    }
}


@Before("check()")
public void before(JoinPoint joinPoint) {
    MethodSignature signature = (MethodSignature) joinPoint.getSignature();
    Method method = signature.getMethod();
    int h;
    int i = ((h = method.getDeclaringClass().hashCode()) ^ (h >>> 16)) & 3;

    Limit limit = method.getAnnotation(Limit.class);
```

```
        if (limit != null) {
            String key = limit.key();
            if (key == null || "".equals(key)) {
                throw new RedisLimitException("key cannot be null");
            }
            StringBuilder limitKey = new StringBuilder();
            limitKey.append(key).append(":")
                    .append(method.getDeclaringClass().getName()).append(":")
                    .append(method.getName()).append(":")
                    .append(limit.addLimit()).append(":").append(limit.incrementPerSecon
d());

            log.info(limitKey.toString());

            long defaultPermits = limit.defaultPermits();
            long startTim = System.currentTimeMillis();
            boolean permit = false;
            while ((System.currentTimeMillis() - startTim) <= limit.waitTimes()) {
                try {
                    Boolean result = stringRedisTemplate.execute(redisScript,
Arrays.asList("funcLimit" + i, limitKey.toString()),
                            String.valueOf(defaultPermits - 1));
```

```
                log.info("Access try result is {} for key={}", result, limitKey);
                if (result != null && !result) {
                    log.debug("令牌桶={}，获取令牌失败", limitKey);
                } else {
                    permit = true;
                    break;
                }
            } catch (Exception e) {
                log.error("拿令牌出错", e);
            }
        }
        if (!permit) {
            throw new RedisLimitException(limit.msg());
        }
        stringRedisTemplate.execute(redisConsumptionScript,
Arrays.asList("funcLimitConsumption" + i, limitKey.toString()));
    }
  }


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

```
public @interface Limit {

    /**
     * 资源的 key,唯一
     * 作用：不同的接口，不同的流量控制
     */
    String key() default "";

    /**
     * 最多的访问限制次数
     */
    long defaultPermits() default 10;

    /**
     * 每次添加的次数
     */
    long incrementPerSecond() default 2;

    /**
     * 等待时间
     */
    long waitTimes() default 0;
```

```
        /**
         * 添加令牌的界限
         */
        long addLimit() default 5;

        /**
         * 得不到令牌的提示语
         */
        String msg() default "系统繁忙,请稍后再试.";
    }
}
```

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD})
@Documented
public @interface RedisLimit {
    /**
     * 资源的 key,唯一
     * 作用：不同的接口，不同的流量控制
```

```
    */
    String key() default "";

    /**
     * 最多的访问限制次数
     */
    long permitsPerSecond() default 2;

    /**
     * 过期时间也可以理解为单位时间，单位秒，默认 60
     */
    long expire() default 60;

    /**
     * 得不到令牌的提示语
     */
    String msg() default "系统繁忙,请稍后再试.";
}
```

```
import com.example.demo3.annotion.RedisLimit;
import com.ldh.utils.exception.RedisLimitException;
import lombok.extern.slf4j.Slf4j;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;
import org.aspectj.lang.reflect.MethodSignature;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.core.io.ClassPathResource;
import org.springframework.data.redis.core.StringRedisTemplate;
import org.springframework.data.redis.core.script.DefaultRedisScript;
import org.springframework.scripting.support.ResourceScriptSource;
import org.springframework.stereotype.Component;
import org.springframework.util.StringUtils;

import javax.annotation.PostConstruct;
import java.lang.reflect.Method;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
```

```
@Slf4j
@Aspect
@Component
public class RedisLimitComponent {

    @Autowired
    private StringRedisTemplate stringRedisTemplate;

    private DefaultRedisScript<Long> redisScript;



    @Pointcut("@annotation(com.example.demo3.annotion.RedisLimit)")
    private void check() {

    }



    @PostConstruct
    public void init(){
        redisScript = new DefaultRedisScript<>();
        redisScript.setResultType(Long.class);
```

```
        redisScript.setScriptSource(new ResourceScriptSource(new
ClassPathResource("lua/rateLimiter.lua")));
    }



    @Before("check()")
    public void before(JoinPoint joinPoint) {
        MethodSignature signature = (MethodSignature) joinPoint.getSignature();
        Method method = signature.getMethod();
        //拿到 RedisLimit 注解，如果存在则说明需要限流
        RedisLimit redisLimit = method.getAnnotation(RedisLimit.class);
        if (redisLimit != null) {
            //获取 redis 的 key
            String key = redisLimit.key();
            String className = method.getDeclaringClass().getName();
            String limitKey = key + className + method.getName();
            log.info(limitKey);
            if (StringUtils.isEmpty(key)) {
                throw new RedisLimitException("key cannot be null");
            }
            long limit = redisLimit.permitsPerSecond();
            long expire = redisLimit.expire();
```

```
            List<String> keys = new ArrayList<>(Arrays.asList(key));
            Long count = stringRedisTemplate.execute(redisScript, keys,
String.valueOf(limit), String.valueOf(expire));
            log.info("Access try count is {} for key={}", count, key);
            if (count != null && count == 0) {
                log.debug("令牌桶={}，获取令牌失败", key);
                throw new RedisLimitException(redisLimit.msg());
            }
        }
    }
}
```

https://github.com/daohonglei/javaStereotypedWriting

https://gitee.com/daohonglei/javaStereotypedWriting