https://mp.weixin.qq.com/s/fk\_rmO9rXu8SAWvHPWZ50A

# 看完这篇Redis缓存三大问题，保你能和面试官互扯。

日常的开发中，无不都是使用数据库来进行数据的存储，由于一般的系统任务中通常不会存在高并发的情况，所以这样看起来并没有什么问题。

一旦涉及大数据量的需求，如一些商品抢购的情景，或者主页访问量瞬间较大的时候，单一使用数据库来保存数据的系统会因为面向磁盘，磁盘读/写速度问题有严重的性能弊端，详细的磁盘读写原理请参考这一片[]。

在这一瞬间成千上万的请求到来，需要系统在极短的时间内完成成千上万次的读/写操作，这个时候往往不是数据库能够承受的，极其容易造成数据库系统瘫痪，最终导致服务宕机的严重生产问题。

为了克服上述的问题，项目通常会引入NoSQL技术，这是一种基于内存的数据库，并且提供一定的持久化功能。

Redis技术就是NoSQL技术中的一种。Redis缓存的使用，极大的提升了应用程序的性能和效率，特别是数据查询方面。

但同时，它也带来了一些问题。其中，最要害的问题，就是数据的一致性问题，从严格意义上讲，这个问题无解。如果对数据的一致性要求很高，那么就不能使用缓存。

另外的一些典型问题就是，**缓存穿透、缓存击穿和缓存雪崩**。本篇文章从实际代码操作，来提出解决这三个缓存问题的方案，毕竟Redis的缓存问题是实际面试中高频问点，理论和实操要兼得。

## 缓存穿透

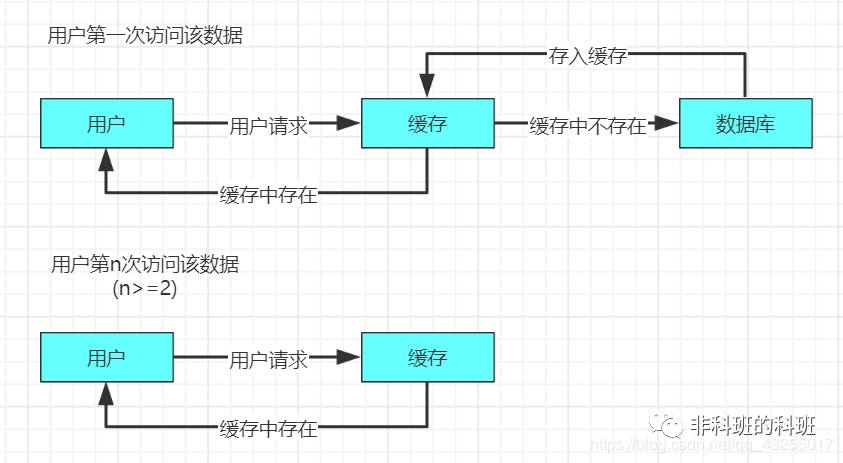
缓存穿透是指查询一条数据库和缓存都没有的一条数据，就会一直查询数据库，对数据库的访问压力就会增大，缓存穿透的解决方案，有以下两种：

* 缓存空对象：代码维护较简单，但是效果不好。
* 布隆过滤器：代码维护复杂，效果很好。

### 缓存空对象

缓存空对象是指当一个请求过来缓存中和数据库中都不存在该请求的数据，第一次请求就会跳过缓存进行数据库的访问，并且访问数据库后返回为空，此时也将该空对象进行缓存。

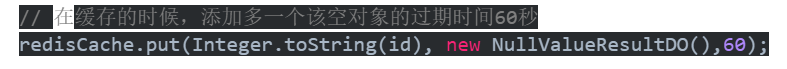
若是再次进行访问该空对象的时候，就会直接击中缓存，而不是再次数据库，缓存空对象实现的原理图如下：



缓存空对象的实现代码如下：



缓存空对象的实现代码很简单，但是缓存空对象会带来比较大的问题，就是缓存中会存在很多空对象，占用内存的空间，浪费资源，一个解决的办法就是设置空对象的较短的过期时间，代码如下：



### 布隆过滤器

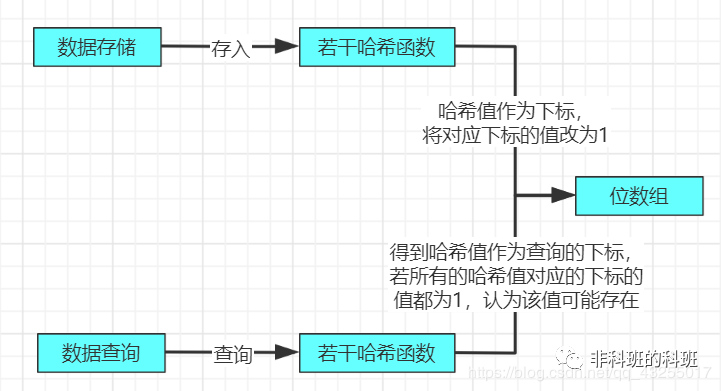
布隆过滤器是一种基于概率的数据结构，主**要用来判断某个元素是否在集合内**，它具有运行速度快（时间效率），占用内存小的优点（空间效率），但是有一定的误识别率和删除困难的问题。它只能告诉你某个元素一定不在集合内或可能在集合内。

在计算机科学中有一种思想：空间换时间，时间换空间。一般两者是不可兼得，而布隆过滤器运行效率和空间大小都兼得，它是怎么做到的呢？

在布隆过滤器中引用了一个误判率的概念，即它可能会把不属于这个集合的元素认为可能属于这个集合，但是不会把属于这个集合的认为不属于这个集合，布隆过滤器的特点如下：

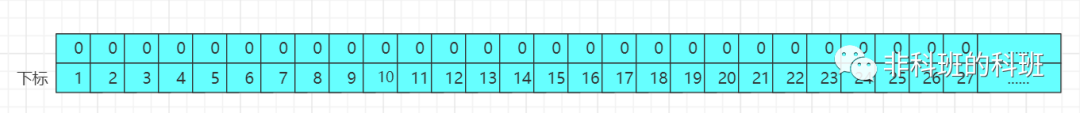
1. 一个非常大的二进制位数组 （数组里只有0和1）
2. 若干个哈希函数
3. 空间效率和查询效率高
4. 不存在漏报（False Negative）：某个元素在某个集合中，肯定能报出来。
5. 可能存在误报（False Positive）：某个元素不在某个集合中，可能也被爆出来。
6. 不提供删除方法，代码维护困难。
7. 位数组初始化都为0，它不存元素的具体值，当元素经过哈希函数哈希后的值（也就是数组下标）对应的数组位置值改为1。

实际布隆过滤器存储数据和查询数据的原理图如下：



可能很多读者看完上面的特点和原理图，还是看不懂，别急下面通过图解一步一步的讲解布隆过滤器，总而言之一句简单的话概括就是布隆过滤器是一个很大二进制的位数组，数组里面只存0和1。

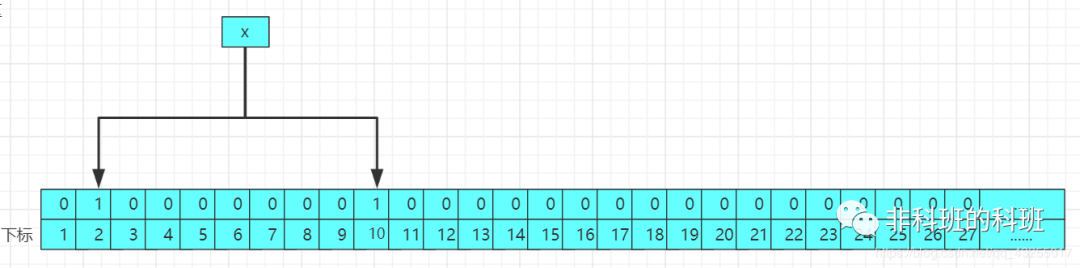
初始化的布隆过滤器的结构图如下：



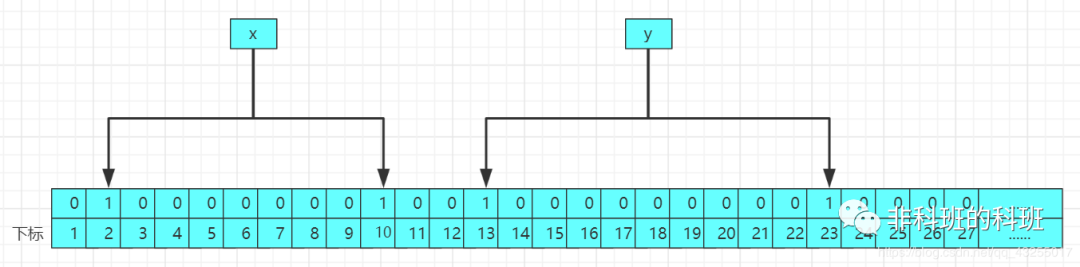
以上只是画了布隆过滤器的很小很小的一部分，实际布隆过滤器是非常大的数组（这里的大是指它的长度大，并不是指它所占的内存空间大）。

那么一个数据是怎么存进布隆过滤器的呢？

当一个数据进行存入布隆过滤器的时候，会经过如干个哈希函数进行哈希（若是对哈希函数还不懂的请参考这一片[]），得到对应的哈希值作为数组的下标，然后将初始化的位数组对应的下标的值修改为1，结果图如下：



当再次进行存入第二个值的时候，修改后的结果的原理图如下：

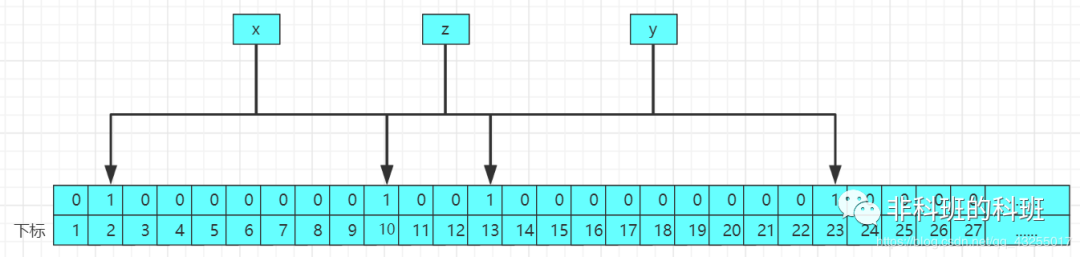


所以每次存入一个数据，就会哈希函数的计算，计算的结果就会作为下标，在布隆过滤器中有多少个哈希函数就会计算出多少个下标，布隆过滤器插入的流程如下：

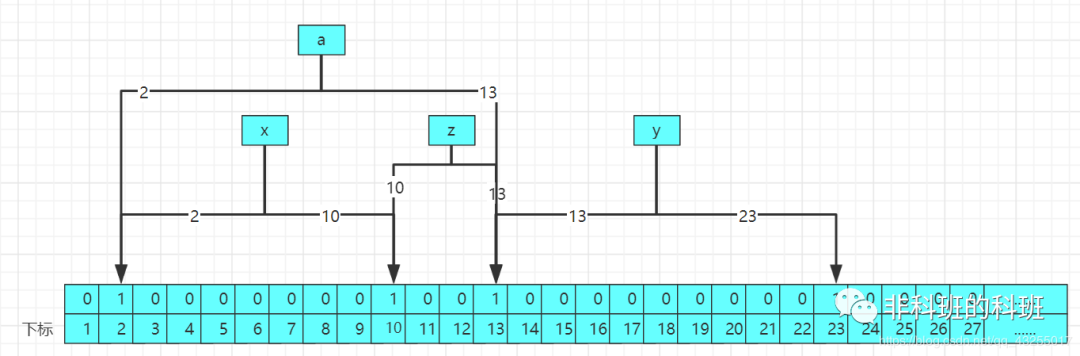
1. 将要添加的元素给m个哈希函数
2. 得到对应于位数组上的m个位置
3. 将这m个位置设为1

那么为什么会有误判率呢？

假设在我们多次存入值后，在布隆过滤器中存在x、y、z这三个值，布隆过滤器的存储结构图如下所示：



当我们要查询的时候，比如查询a这个数，实际中a这个数是不存在布隆过滤器中的，经过2个哈希函数计算后得到a的哈希值分别为2和13，结构原理图如下：



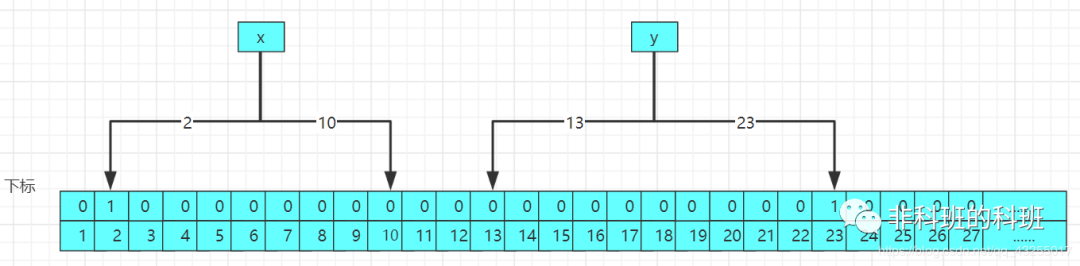
经过查询后，发现2和13位置所存储的值都为1，但是2和13的下标分别是x和z经过计算后的下标位置的修改，该布隆过滤器中实际不存在a，那么布隆过滤器就会误判改值可能存在，因为布隆过滤器不存元素值，所以存在误判率。

那么具体布隆过布隆过滤的判断的准确率和一下两个因素有关：

* 布隆过滤器大小：越大，误判率就越小，所以说布隆过滤器一般长度都是非常大的。
* 哈希函数的个数：哈希函数的个数越多，那么误判率就越小。

那么为什么不能删除元素呢？

原因很简单，因为删除元素后，将对应元素的下标设置为零，可能别的元素的下标也引用改下标，这样别的元素的判断就会收到影响，原理图如下：



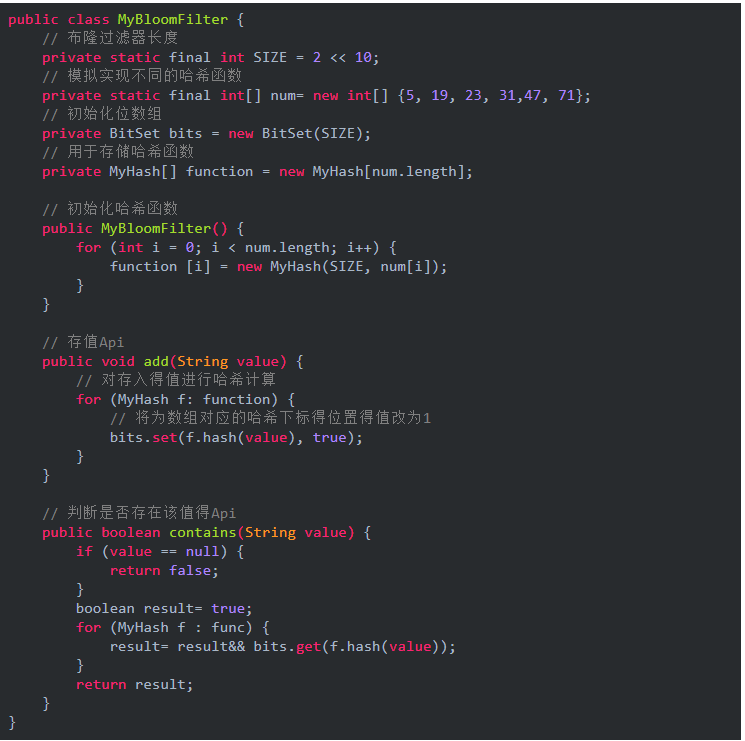
当你删除z元素之后，将对应的下标10和13设置为0，这样导致x和y元素的下标受到影响，导致数据的判断不准确，所以直接不提供删除元素的api。

以上说的都是布隆过滤器的原理，只有理解了原理，在实际的运用才能如鱼得水，下面就来实操代码，手写一个简单的布隆过滤器。

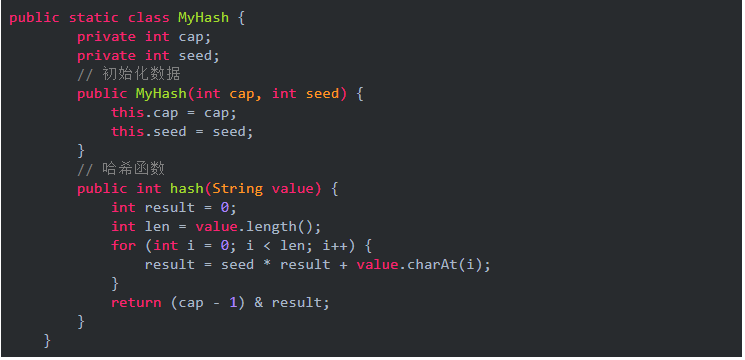
对于要手写一个布隆过滤器，首先要明确布隆过滤器的核心：

* 若干哈希函数
* 存值的Api
* 判断值的Api

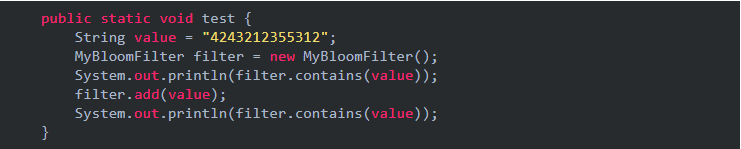
实现的代码如下：



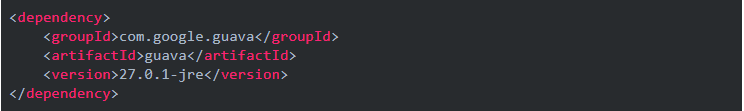
哈希函数代码如下：



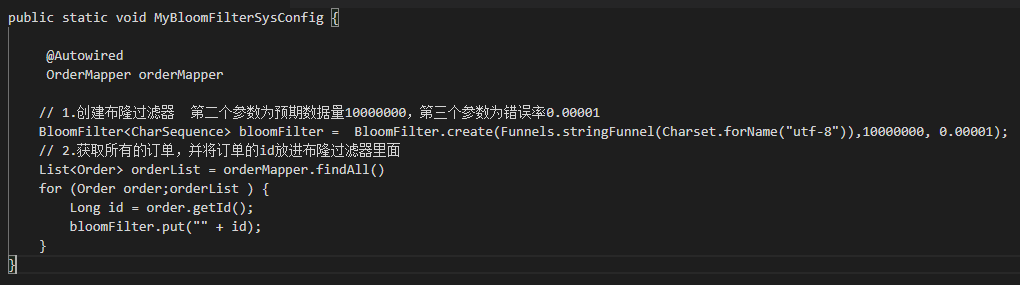
布隆过滤器测试代码如下：



以上就是手写了一个非常简单的布隆过滤器，但是实际项目中可能由牛人或者大公司已经帮你写好的，如谷歌的Google Guava，只需要在项目中引入一下依赖：



实际项目中具体的操作代码如下：



在实际项目中会启动一个**系统任务**或者**定时任务**，来初始化布隆过滤器，将热点查询数据的id放进布隆过滤器里面，当用户再次请求的时候，使用布隆过滤器进行判断，改订单的id是否在布隆过滤器中存在，不存在直接返回null，具体操作代码：



布隆过滤器的缺点就是要维持容器中的数据，因为订单数据肯定是频繁变化的，实时的要更新布隆过滤器中的数据为最新。

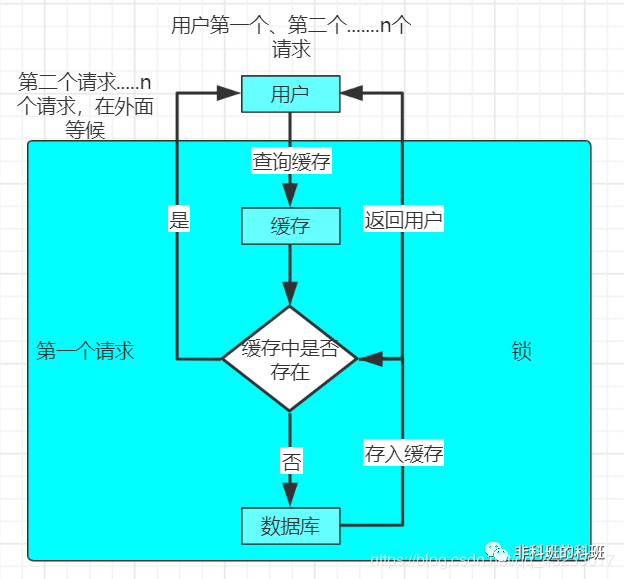
## 缓存击穿

缓存击穿是指一个key非常热点，在不停的扛着大并发，大并发集中对这一个点进行访问，当这个key在失效的瞬间，持续的大并发就穿破缓存，直接请求数据库，瞬间对数据库的访问压力增大。

缓存击穿这里强调的是并发，造成缓存击穿的原因有以下两个：

* 该数据没有人查询过 ，第一次就大并发的访问。（冷门数据）
* 添加到了缓存，reids有设置数据失效的时间 ，这条数据刚好失效，大并发访问（热点数据）

对于缓存击穿的解决方案就是加锁，具体实现的原理图如下：



当用户出现**大并发**访问的时候，在查询缓存的时候和查询数据库的过程加锁，只能第一个进来的请求进行执行，当第一个请求把该数据放进缓存中，接下来的访问就会直接集中缓存，防止了**缓存击穿**。

业界比价普遍的一种做法，即根据key获取value值为空时，锁上，从数据库中load数据后再释放锁。若其它线程获取锁失败，则等待一段时间后重试。这里要注意，分布式环境中要使用**分布式锁**，**单机**的话用普通的锁（synchronized、Lock）就够了。

下面以一个获取商品库存的案例进行代码的演示，**单机版**的锁实现具体实现的代码如下：



**分布式**的锁实现具体实现的代码如下：



## 缓存雪崩

缓存雪崩 是指在某一个时间段，缓存集中过期失效。此刻无数的请求直接绕开缓存，直接请求数据库。

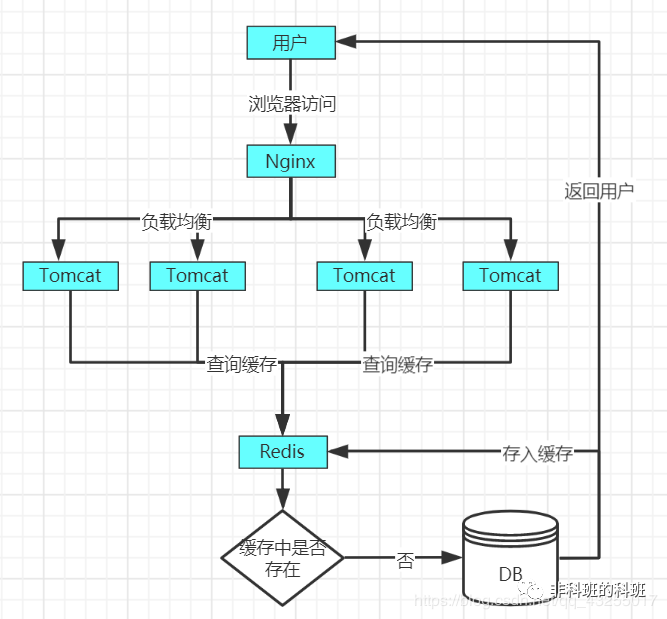
造成缓存雪崩的原因，有以下两种：

* reids宕机
* 大部分数据失效

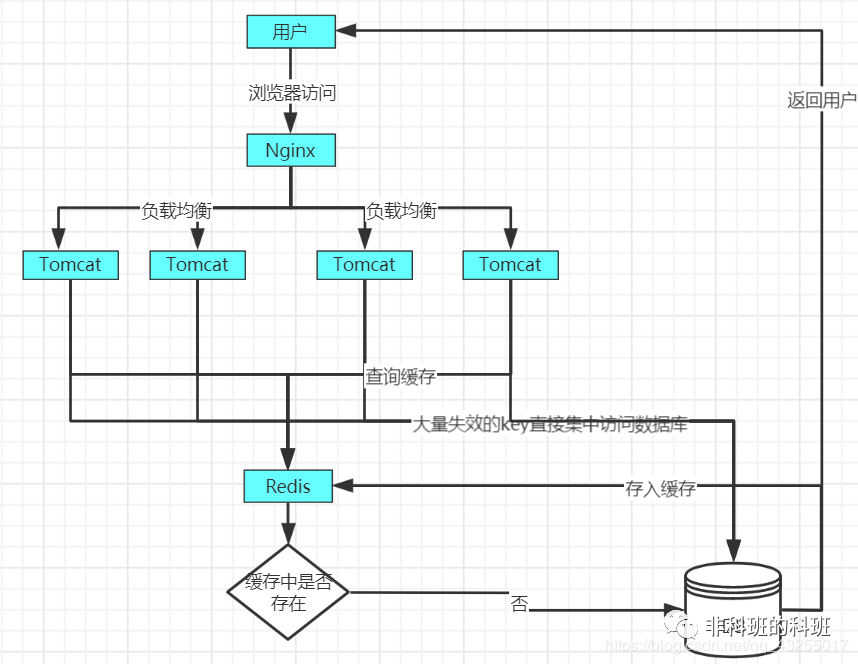
比如天猫双11，马上就要到双11零点，很快就会迎来一波抢购，这波商品在23点集中的放入了缓存，假设缓存一个小时，那么到了凌晨24点的时候，这批商品的缓存就都过期了。

而对这批商品的访问查询，都落到了数据库上，对于数据库而言，就会产生周期性的压力波峰，对数据库造成压力，甚至压垮数据库。

缓存雪崩的原理图如下，当正常的情况下，key没有大量失效的用户访问原理图如下：



当某一时间点，key大量失效，造成的缓存雪崩的原理图如下：



对于缓存雪崩的解决方案有以下两种：

* 搭建高可用的集群，防止单机的redis宕机。
* 设置不同的过期时间，防止同一时间内大量的key失效。

针对业务系统，永远都是具体情况具体分析，没有最好，只有最合适。于缓存其它问题，缓存满了和数据丢失等问题，我们后面继续深入的学习。最后也提一下三个词LRU、RDB、AOF，通常我们采用LRU策略处理溢出，Redis的RDB和AOF持久化策略来保证一定情况下的数据安全。