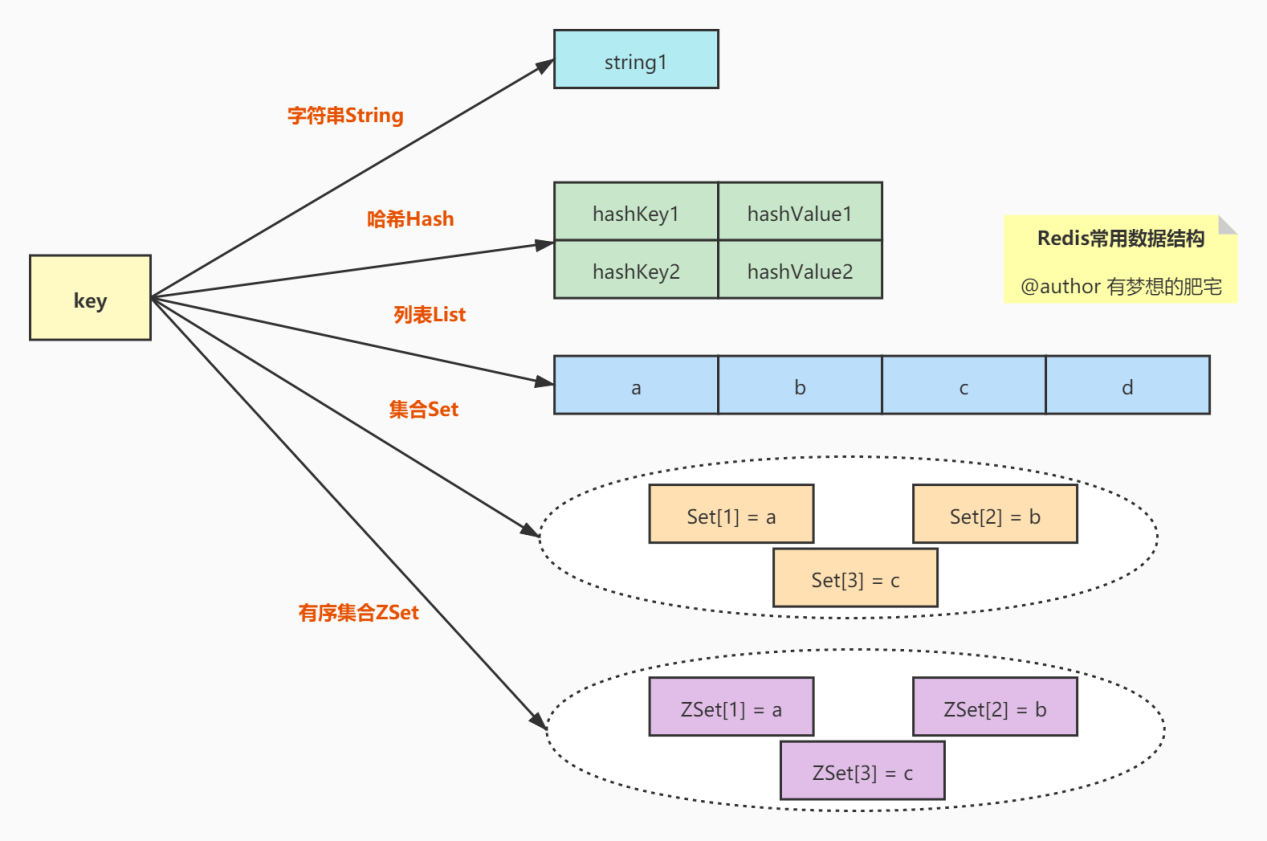
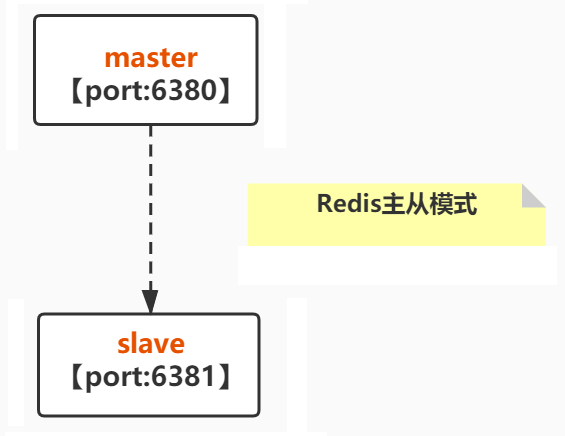
## 1.Redis



常用数据结构

### Redis主从架构



**主从同步流程：**

1、当我们为master配置了一个slave，不管这个slave是否是第一次连接上Master，它都会发送一个PSYNC命令给master请求复制数据。

2、master收到PSYNC命令后，会在后台进行数据持久化通过bgsave生成最新的rdb快照文件

3、持久化期间，master会继续接收客户端的请求，它会把这些可能修改数据集的请求缓存在内存中。

4、当持久化进行完毕以后，master会把这份rdb文件数据集发送给slave，slave会把接收到的数据进行持久化生成rdb，然后再加载到内存中。

5、master再将之前缓存在内存中的命令发送给slave。

PS：当master与slave之间的连接由于某些原因而断开时，slave能够自动重连Master，如果master收到了多个slave并发连接请求，它只会进行一次持久化，而不是一个连接一次，然后再把这一份持久化的数据发送给多个并发连接的slave。

PS：SYNC使用的是全量复制，PSYNC使用的是增量复制。

PS：主从结构一般会设置成：主节点负责写入，从节点负责备份和读取。

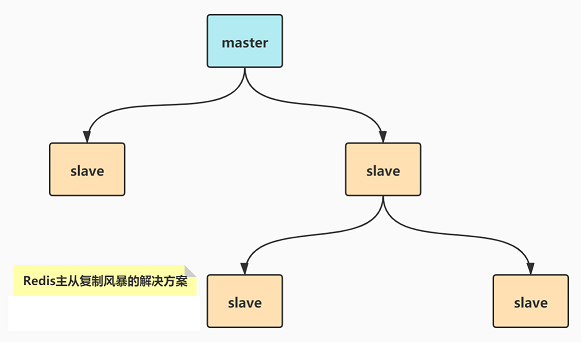
PS：主从同步与开不开启持久化无关，是redis服务端做的事情。至于为什么用rdb文件来同步也是因为rdb是二进制形式的数据，同步起来比aof更快。

**Redis主从复制风暴的解决方案：**

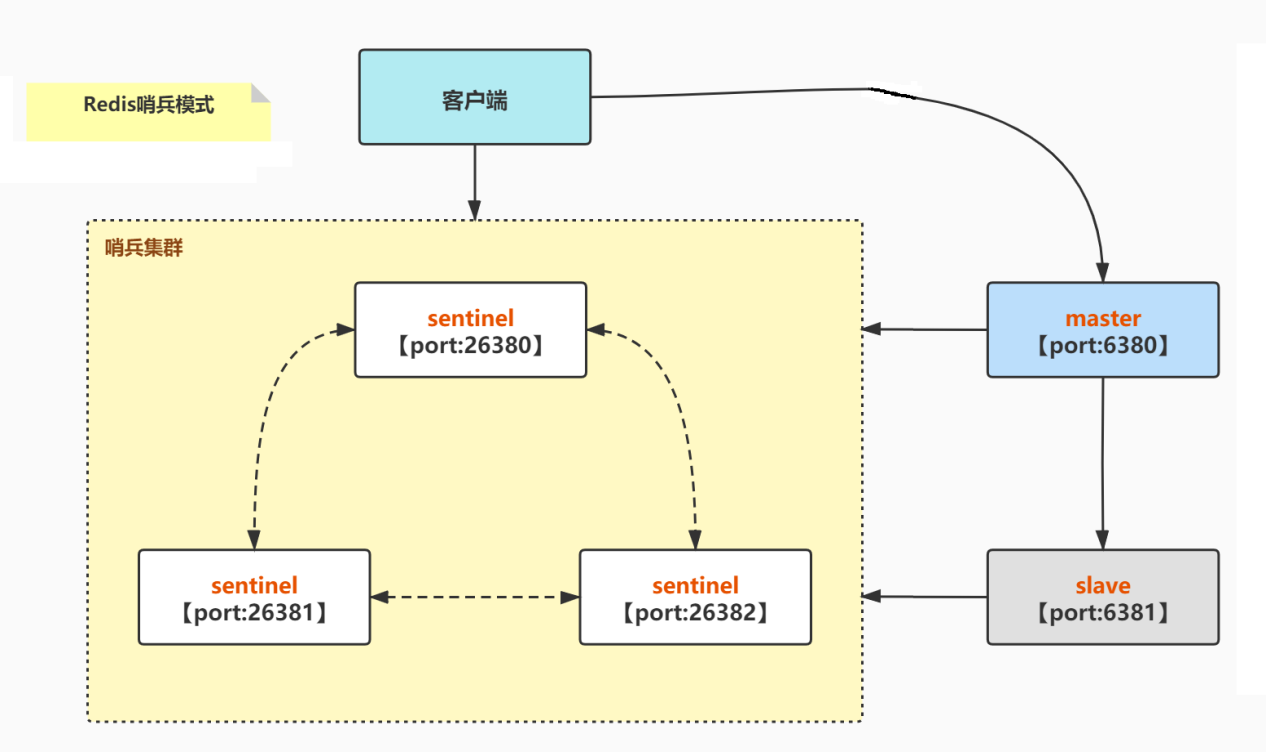
主从复制风暴：多个从节点同时复制主节点导致主节点压力过大。

解决方案：让部分从节点向另一些从节点同步数据。PS：这里被同步的从节点要持续与主节点进行同步。

PS：主节点一般配置不超过10个G因为太大了会影响同步时的效率



### Redis哨兵架构



1.sentinel哨兵是特殊的redis服务，不提供读写服务，主要用来监控redis实例节点。

2.哨兵架构下客户端第一次需要从哨兵找出redis的主节点

3.后续就直接访问redis的主节点，不会每次都通过 sentinel代理访问redis的主节点

4.当redis的主节点发生变化，哨兵会第一时间感知到，并且将新的redis主节点通知给客户端。

PS：这里面redis的客户端一般都实现了订阅功能，订阅sentinel发布的节点变动消息

**哨兵概念**

　　哨兵其实也是redis实例，只不过不提供读写服务，主要用来监控redis实例节点。

　　哨兵一般配单数（半数选举机制）

**半数选举机制**

半数选举机制主要指的是：

1、当哨兵监控的主节点挂了，那么必须有一半以上的哨兵认为当前主节点挂了，才会真正开始选举新的主节点。

2、当开始选举新的主节点时，从节点必须得到一半以上的哨兵“投票”，才能成为新的主节点。

　　PS：哨兵全部挂了，则集群瘫痪，无法访问。

**访问瞬断**

　　由于网络波动或者其他原因发生网络访问瞬断时，redis会开始检测主节点信息，如果通过半数确认redis挂了，则哨兵会去选举新的主节点，在选举期间redis无法提供服务。

**主节点访问问题**

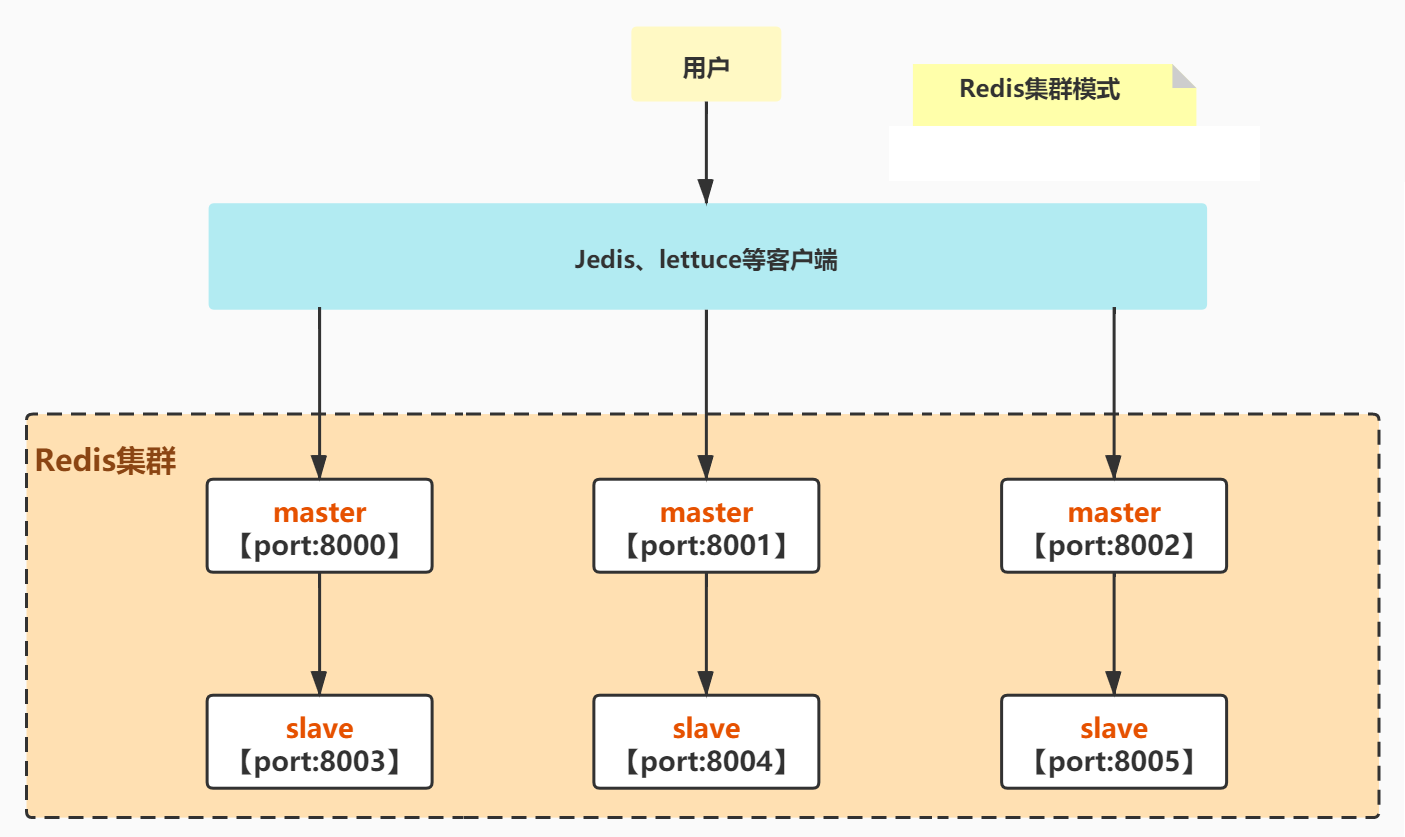
由于哨兵模式下从节点会去拉取主节点的数据，主节点一般配置不超过10个G，因为太大了会影响同步时的效率。

**Redis哨兵选举原理分析**

* 1、当一个master服务器被某sentinel视为下线状态后，该sentinel会与其他sentinel协商**选出sentinel的leader**进行故障转移工作**【相当于sentinel的master】**。
* 2、每个发现master服务器进入下线的sentinel都可以要求其他sentinel选自己为sentinel的leader，选举是先到先得**【sentinel开始竞争当leader~】**。
* 3、同时**每个sentinel每次选举都会自增配置纪元(选举周期)【currentEpoch】**，每个纪元中只会选择一 个sentinel的leader**【每次选举只有一个Leader~】**。
* 4、如果所有超过一半的sentinel选举某sentinel作为leader。之后该sentinel进行故障转移操作，从存活的slave中选举出新的master**【sentinel的leader去给slave选出新的老大当master~】**。
* 5、如果哨兵集群只有一个哨兵节点，redis的主从也能正常运行以及选举master，如果master挂了，那唯一的那个哨兵节点就是哨兵leader了，也可以正常选举新master**【只有一个sentinel那就不用选了，它自己就是leader】**。

**PS：**为了高可用一般都推荐至少部署三个哨兵节点。【奇数原理会在后面的集群篇重点解释，先有一个概念】

### Redis集群架构

****

* Redis集群是一个**由多个主从节点群组成的分布式服务器群**，它具有**复制**、**高可用**和**分片**特性。
* Redis集群不需要sentinel哨兵也能完成节点移除和故障转移的功能。
* 需要将每个节点设置成集群模式，这种集群模式**没有中心节点**，**可水平扩展**。**PS：**官方推荐不超过1000个节点。
* redis集群的性能和高可用性均优于之前版本的哨兵模式，且集群配置非常简单。

**Redis集群原理的一些概念**

Redis Cluster 将所有数据划分为**16384**slots(槽位)，**每个节点负责其中一部分槽位**。槽位的信息存储于每个节点中。

　　当 Redis Cluster 的客户端来连接集群时，它也会得到一份**集群的槽位配置信息并将其缓存在客户端本地**。这样当客户端要查找某个key时，可以直接定位到目标节点。

**PS：**因为槽位的信息可能会存在客户端与服务器不一致的情况，还需要纠正机制来实现槽位信息的校验调整。

槽位定位算法

Cluster 默认会对key值使用**crc16算法进行hash得到一个整数值**，然后用这个整数值对16384进行取模来得到具体槽位。

**HASH\_SLOT = CRC16(key) mod 16384**

**PS：**新节点加入集群中时是无法写数据的（没有slot槽位），需要从新分配槽位后才可以访问

**PS：**迁移槽位会把对应的数据也迁移过去

**PS：**新节点加入集群时都是master节点

跳转重定位

当客户端向一个错误的节点发出了指令，该节点会**发现指令的key所在的槽位并不归自己管理，这时它会向客户端发送一个特殊的跳转指令携带目标操作的节点地址，告诉客户端去连这个节点去获取数据**。

客户端收到指令后除了跳转到正确的节点上去操作，还会**同步更新纠正本地的槽位映射表缓存，后续所有key将使用新的槽位映射表**。

Redis集群节点间的通信机制

Redis cluster节点间采取**gossip协议**进行通信，有两种方式：

集中式

**优点：**元数据的更新和读取，**时效性非常好**，一旦元数据出现变更立即就会更新到集中式的存储中，其他节点读取的时候立即就可以立即感知到。

**缺点：所有的元数据的更新压力全部集中在一个地方**，可能导致元数据的**存储压力**。

**PS：**很多中间件都会借助zookeeper集中式存储元数据。

**gossip**

**优点：**元数据的**更新比较分散**，不是集中在一个地方，更新请求会陆陆续续，打到所有节点上去更新，**有一定的延时，降低了压力**。

**缺点：**元数据**更新有延时可能导致集群的一些操作会有一些滞后**。

gossip协议包含多种消息，包括ping，pong，meet，fail等等。

**meet：**某个节点发送meet给新加入的节点，**让新节点加入集群中**，然后新节点就会开始与其他节点进行通信。

**ping：**每个节点都会频繁给其他节点发送ping，其中包含自己的状态还有自己维护的集群元数据，互相通过ping**交换元数据【类似自己感知到的集群节点增加和移除，hash slot信息等】**。

**pong：对ping和meet消息的返回**，包含自己的状态和其他信息，也可以用于信息广播和更新。

**fail：**某个节点判断另一个节点fail之后，就发送fail给其他节点，**通知其他节点，指定的节点宕机了**。

gossip通信的10000端口

每个节点都有一个专门用于节点间gossip通信的端口，就是**自己提供服务的端口号+10000**，

**举个🌰：**比如8000，那么用于节点间通信的就是18000端口。

网络抖动

**定义：**网络通信中，有时会因为一些客观原因使得网络突然中断或者波动而变得不可访问。

Redis Cluster 提供了一种选项**cluster-­node-­timeout**，表示**当某个节点持续timeout的时间失联时，才可以认定该节点出现故障，需要进行主从切换**。

**PS：**这个配置很重要，**如果没有这个选项，网络抖动会导致主从频繁切换** **【数据的重新复制】**。

Redis集群选举原理分析

当slave发现自己的master变为FAIL状态时，便尝试进行Failover，以期成为新的master。

　　由于挂掉的master可能会有多个slave，从而存在多个slave竞争成为master节点的过程，其过程如下：

1、slave发现自己的master变为**FAIL**

2、将自己记录的集群currentEpoch【当前分代】加1，并广播**FAILOVER\_AUTH\_REQUEST**信息

3、其他节点收到该信息，**只有master响应**，判断请求者的合法性，并发送**FAILOVER\_AUTH\_ACK【相当于投票】**。**PS：**对每一个epoch只发送一次ack。

4、尝试**Failover【期待上位~】**的slave收集master返回的**FAILOVER\_AUTH\_ACK【投的票数】**

5、slave收到超过半数master的ack后变成新Master**【半数选举机制】**(这里解释了集群为什么至少需要三个主节点，如果只有两个，当其中一个挂了，只剩一个主节点是不能选举成功的)

6、slave广播**Pong**消息通知其他集群节点。

Redis集群脑裂问题

问题的产生

一个小集群内部主从因网络问题**无法同步数据**导致了**重新选举【PS：这时主节点是存活的，只是网络问题导致无法同步】**而后网络恢复导致该小集群**出现2个主节点对外提供服务**。

规避的方法

在redis.conf配置文件下更改：

min‐replicas‐to‐write 1 #这个配置的意思是主节点写数据完成后，最少需要同步的slave数量【举个例子：比如1主2从的机制，原来的主节点因为网络波动被判定为挂了，此时新的从节点想上位，那么它必须要能给另外1个从节点进行redis写的同步，它才能成为主节点。此时对于这个从节点除了自己以外他还有一个从节点对其进行数据同步，那么1+1>2/3符合半数选举机制】

**PS：**这个配置在一定程度上**会影响集群的可用性**，比如slave要是少于1个，那么没有从节点写入数据，就满足不了这个配置，那就不能对外提供服务了。

### Redis集群和哨兵的区别

1、**哨兵**只提供一个主节点，同步时也是全量同步。**集群**的数据是分片放的，分片意味着数据是不重叠的。

2、**哨兵**架构可以写主读从，但是**集群**架构读写都走主节点。

3、**哨兵**架构收到单机数据量制约【一般不超过10个G】，**集群**架构可以很方便的进行水平扩展【当然操作起来还需要分配slots槽等等是有些麻烦】。

4、**哨兵**架构中的哨兵角色只是起到监控的作用，但不对外提供服务，相当于占用了资源。**集群**的话每一台主机都是主/或从，就不会有哨兵角色的这种主机资源“浪费”。

### Redis的QA

Q：Redis集群为什么至少需要三个master节点，并且推荐节点数为奇数？

因为新master的选举需要大于半数的集群master节点同意才能选举成功，如果只有两个master节点，当其中 一个挂了，是达不到选举新master的条件的。

**举个🌰：**3个master，挂了1个，则挂了的master其对应的slave若是获得了另外两个master的支持【2>3/2】，则它就可以成为master。

**举个反🌰：**2个master，挂了1个，那么就算另外一个master选出一个slave作为新的master，此时票数也不够【1=2/2】，达不到半数效果。

**举个反🌰：**4个master，如果挂了1个，选举方式和3个master的方式类似，可以达到半数选举的效果。但如果挂了2个，场景和上面的类似，也一样无法选出新的master。

**小结**

**为什么至少需要3个master节点：半数选举机制**

**为什么推荐节点数为奇数：**从**节省机器资源角度出发考虑**的

Q：集群是否完整才能对外提供服务？

不一定，redis.conf可以配置：

**#表示当负责一个插槽的主库下线且没有相应的从库进行故障恢复时，集群仍然可用，如果为yes则集群不可用。**

cluster-require-full-coverage no

Q：为啥redis.cond文件内部要写集群信息？

因为当关闭所有机器的时候，再次重启**可以通过配置文件来恢复集群信息**。

**PS：集群重启不要用创建集群的命令，直接启动单台机器即可。**

Q：集群内部主从为啥不放在一台机器上？

**为了更安全**，如果放在一台机器上，一个小集群挂了就整个集群全部挂掉。

### Redis使用问题

**缓存穿透**

**定义：查询一个根本不存在的数据**，则缓存层和存储层都不会命中。

**弊端：**缓存穿透将导致不存在的数据每次请求都要到存储层去查询，失去了缓存保护后端存储的意义。

造成缓存穿透的基本原因有两个：

1、自身业务代码或者数据出现问题。

2、一些恶意攻击、爬虫等造成大量缓存空命中。

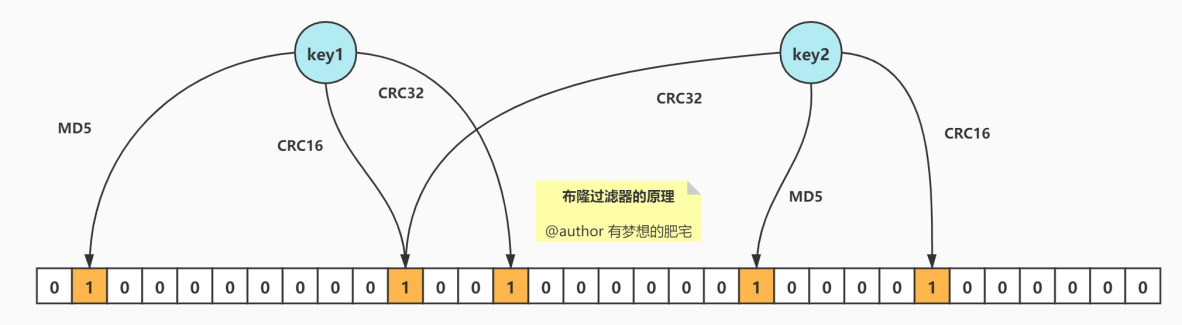
布隆过滤器

**定义：**布隆过滤器就是**一个大型的位数组和几个不一样的无偏hash函数**。

**PS：**无偏就是能够把元素的**hash值算得比较均匀**。

**使用方式：redisson**内部对布隆过滤器做了整合，需要使用的时候在工程引入redisson依赖就好

**含义：当布隆过滤器说某个值存在时，这个值可能不存在；当它说不存在时，那就肯定不存在**。



**解析：**

1、布隆过滤器会**对key进行一些hash运算，求出几个hash位置**，然后会在布隆过滤器里面维护的**大型的位数组**中，**把对应的位置设置为1**。

2、那么当我们下次用key来访问时，布隆过滤器也会对其运算，然后查看对应的槽位中的值是不是为1，如果都为1则布隆过滤器返回存在，如果其中1个不为1，则返回key不存在。

**PS：**从图上我们可以看到为什么布隆过滤器返回key不存在时，是一定不存在的。但是当布隆过滤器返回存在时，key却不一定真实存在，因为可能会有hash碰撞。

**PS：**虽然这个位数组长度很大，但其实**占用的内存很小**，因为是**位数组**，8位才1个字节。

**PS：**布隆过滤器不能删除数据，如果要删除得重新初始化数据。

**PS：**布隆过滤器主要是为了在高并发的场景下做一层防御防止缓存击穿的问题，**其并不能保证结果是100%准确的**。【剩余的漏网之鱼请求下沉到数据库也问题不大了，因为布隆过滤器很强大，已经能帮我们挡掉绝大部分的缓存击穿请求了】

**缓存失效**

**定义：**由于**大批量缓存在同一时间失效**可能导致大量请求同时穿透缓存直达数据库，可能会造成数据库瞬间压力过大甚至挂掉。

解决方案

将缓存过期时间设置为一个时间段内的不同时间

**缓存雪崩**

**定义：**由缓存雪崩指的是**redis集群因为设计或者请求量大等原因挂了，请求会穿透到db层把数据库也打挂掉，从而引起整个系统的瘫痪**。

预防和解决

1、保证缓存层服务高可用性，比如使用[哨兵模式](https://www.cnblogs.com/riches/p/15095445.html)或[集群模式](https://www.cnblogs.com/riches/p/15106238.html)。

2、使用Sentinel或Hystrix限流降级组件。

3、项目上线前的压测，以及宕机后的预案整备。

**热点缓存key重建优化**

考虑了上面几种情况的解决方案时，redis基本已经可以保护我们的系统了，但是如果出现下面两种情况，可能又会把我们的系统搞挂...

1. 一个突然的秒杀活动或者明星效应使得一个**平时不怎么访问的商品突然访问量暴增**，使其变成**热点key**
2. 重建缓存比较复杂，重建过程耗时比较久。

那么在缓存失效的瞬间，如果有大量线程来重建缓存，会造成后端压力急速增加，甚至可能会让系统崩溃。

解决方案：Redis分布式锁来控制重建缓存的过程

**缓存与数据库双写不一致**

**解决方案**

1、对于并发几率很小的数据(如个人维度的订单数据、用户数据等)，这种几乎不用考虑这个问题，很少会发生缓存不一致，可以给缓存数据加上过期时间，每隔一段时间触发读的主动更新即可。

2、就算并发很高，如果业务上能容忍短时间的缓存数据不一致(如商品名称，商品分类菜单等)，缓存加上过期时间依然可以解决大部分业务对于缓存的要求。

3、如果不能容忍缓存数据不一致，可以通过加**读写锁**保证并发读写或写写的时候按顺序排好队，**读读的时候相当于无锁**。**【Redisson对读写锁也做了封装】**

4、可以用阿里开源的**canal**通过**监听数据库的binlog日志及时的去修改缓存**，但是引入了新的中间件，增加了系统的复杂度。

**小结**

**读多写少：**可以**加入Redis中间件提高性能**。

**读多写多且需要保证数据高度一致：直接操作数据库，不建议使用Redis**。

**PS：**放入缓存的数据应该是对**实时性、一致性要求不是很高的数据**。

**PS：**不要为了用缓存，同时又要保证绝对的一致性做大量的过度设计和控制，增加系统复杂性！

**Redis的过期删除策略**

**策略类型：**

1、**被动删除/惰性删除：**key过期以后不会立即删除，当下一次访问这个key时，才会删除。

2、**主动删除：**由于惰性删除策略无法保证冷数据被及时删掉，所以Redis会定期主动淘汰一批已过期的key。

3、 当前已用内存超过**maxmemory**限定时，触发**主动清理策略**。

**PS：只有主节点才会执行过期删除策略**，主节点删除完了以后会同步**del key**命令给从节点。

主动删除的几种方式

针对设置了过期时间的key做处理

1、**volatile-ttl：**在筛选时，会**针对设置了过期时间的键值对，根据过期时间的先后进行删除**，越早过期的越先被删除。

2、**volatile-random：**就像它的名称一样，在设置了过期时间的键值对中，进行随机删除。

3、**volatile-lru：**会使用 LRU 算法筛选设置了过期时间的键值对删除。

4、**volatile-lfu：**会使用 LFU 算法筛选设置了过期时间的键值对删除。

**针对所有的key做处理**

5、**allkeys-random：**从所有键值对中随机选择并删除数据。

6、**allkeys-lru：**使用 LRU 算法在所有数据中进行筛选删除。

7、**allkeys-lfu：**使用 LFU 算法在所有数据中进行筛选删除。

**不处理**

8、**noeviction：**不会剔除任何数据，**拒绝所有写入操作并返回客户端错误信息**"(error) **OOM** command not allowed when used memory"，**此时Redis只响应读操作**。两种删除算法

LRU 算法【最近最少使用】

淘汰很久没被访问过的数据，以**最近一次访问时间**作为参考。

LFU 算法【最不经常使用】

淘汰最近一段时间被访问次数最少的数据，以**访问次数**作为参考。