# Redis

**NoSql**

用于存储非关系型数据,如json,地理位置，图结构，键值对等

最终一致性

牺牲了强一致性来换取了高性能和高可用

键值存储

文档存储：MongoDB， 存储类似JSON的文档

列族存储

图数据库：节点、关系、属性存储，处理好友关系/知识图谱

**Redis过期删除策略**

Redis的过期删除策略：惰性删除 + 定期删除配合使用

定期删除：定时检查key是否过期

惰性删除：key过期后不操作，下次用到key时再检查，减少检查次数，key过期，但一直未使用，数量越多，占用内存越多;

**为什么用redis缓存而不用本地缓存**

本地缓存无法共享；

每个本地缓存都独立，可能数据不一致，更新完数据库后要更新每个本地缓存

无法持久化

Redis内存更大，支持原子操作，事务，分布式锁等

**布隆过滤器**

二进制向量表(1表示存在，0表示不存在）

**防止缓存穿透**

请求中的key大量不存在时(如：恶意攻击)

1.redis中存null值，相同key请求时，redis就能返回(无效key越多，占用内存越多)

2.将key放入布隆过滤器，先用布隆过滤判断key是否存在,不存在则直接返回

**防止缓存击穿**

redis中热点数据失效，大量请求到达数据库，压垮数据库

1.用互斥锁保证只有一个线程访问数据库，其他线程等待，并将结果写入缓存

2.热点数据快失效时，提前异步更新缓存，延长缓存时间

3.热点数据用不失效

4.热点数据和普通数据存在不同服务器，热点服务器宕机也不影响其他正常请求

**防止缓存雪崩**

Redis中大量key同时过期，大量请求到达数据库，压垮数据库

方案1：key过期时间+随机数（如随机1-5分钟），防止key同时过期

方案2：用队列/锁限制数据库查询,控制查询线程数，降低并发

方案3：双key策略，主key设置过期时间，备key不设置过期时间，当主key失效时，直接返回备key值

**BitMap是什么**

一个大数组，用bit位表示该元素是否存在的value，0表示不存在，1表示存在

一个int为4字节 = 4\*8 = 32个bit位

20亿的int存储需要200000000\*4/1024/1024/1024 = 7.45G

20亿用bit位来表示int是否存在需要 200000000/8/1024/1024/1024 = 0.233G

优点：可以大大减少存储空间

**布隆过滤器是什么**

一个数组，默认全是0，计算数据的哈希值，在相应的数组位置上存1，

查询时，计算哈希值，相应的数组位置为1则存在，为0则不存在。

如果判定元存在，由于哈希碰撞，该元素不一定会存在

如果判定元素不存在，该元素一定不存在（将一定不存在的查询拦截掉）

**redis中zset为什么使用跳表而不用红黑树/平衡树**

跳表实现简单，不用旋转节点

跳表在范围查询时效率高于红黑树，从上往下找，快速定位查询范围。

平衡树的插入/删除会导致子树调整，跳表只修改相邻节点指针

**为什么redis中要使用I/O多路复用**

IO多路复用处理IO请求，避免阻塞，提高访问速度

**redis如何提高缓存命中率**

缓存使用场景:读多写少，时效性低，高并发

提高缓存命中率：颗粒度越小，命中率越高（缓存频繁访问的信息，而不是整个用户信息），避免频繁更新缓存

设置合理的过期时间，避免频繁失效

**redis如何进行模糊查询**

\*：匹配多个字符: keys o\* 查询到'one'

？:匹配单个字符: keys on? 查询到到'one'

[]:通配括号内的某一个字符：keys on[eaw] 查询到‘one’

**Redis中有1亿个key，10w个key以user开头，如何全部找出**

用keys指令， keys user\* 进行模糊查询

**keys和scan的区别**

Keys会阻塞redis主线程，一次返回所有匹配的key。会阻塞服务器，导致其他查询变慢

Scan不阻塞redis主线程, 分批匹配key,每个批次之间也会阻塞线程，当批大小等于key的总量时(等同于keys)，会有一定的重复概率，需要进行去重

**redis缓存预热**

防止服务器启动后，大量请求直接到数据，导致压垮数据库，

提前将热点数据通过脚本方式加入到redis中，再启动服务，避免(穿透，击穿，雪崩)

**百亿量级数据如何存放到redis中**

保留热数据，淘汰冷数据(不常用的数据)

**作缓存的话，缓存key是什么，怎么设计的，什么业务场景？体量规模是多少**

缓存key是业务单元Code，通过key来上锁，防止同时修改

**redis 中key的方式有哪些**

用keys、scan命令模糊匹配key

用Jedis,redisson依赖包，代码中调用keys/scan获取key

用可视化工具，如QuickRedis来查询key

**为什么Redis可以做分布式锁**

所有服务器都访问同一个redis

Redis的操作都是原子性的

setnx命令：只有key不存在时，才设置key值，返回true(幂等性)；

并发时，只有一个线程能访问成功，并插入键值对(原子性)

**Redis 集群主从复制**

从节点启动时，向主节点发送sync命令，主节点收到sync后保存rbd快照,保存期间收到的命令缓存起来

保存完后，主节点将快照和缓存的命令发送给从节点，初始化完成。

之后主节点会将收到的每个命令都发送给从节点。

当断开重连后，2.8之后的版本会将断线期间的命令传给从节点。增量复制

**哨兵模式**

监控redis主、从节点是否正常运行，当主节点出现故障自动将从节点转换为主节点

哨兵至少要 3 个节点，保证健壮性

哨兵 + redis 的主从架构，不保证数据零丢失，只保证 redis 集群的高可用性

**分片模式**

将数据分散到多个Redis节点上，突破单机内存和性能限制

哈希求余算法

一致性哈希算法

哈希槽分区算法

**红锁**

场景：请求分布式锁成功后，从节点还没复制锁，主节点宕机后，应用继续请求锁时，会新的主节点上申请，也会成功。

方案：redis节点相互独立，没有主从之分；申请分布式锁时向所有节点申请(插入key)，当半数申请成功后，才算获得锁。如果没超半数，就要向申请成功的节点删除对应的key.降低问题发生的概率

**redis使用场景**

缓存：

分布式锁 setnx

存储登录信息： hash中key为用户id, value为用户信息

计数器：原子操作自增+1

限流：秒杀场景（SET、EXPIRE 和 Lua 脚本实现）

排行榜：有序集合

Redis分页：list中lrange

**Redis操作为什么是原子的**

redis是单线程操作内存数据，单条命令是原子的，

通过事务保证批量命名操作是原子的

**保证Redis 和 Mysql 的数据一致性用的是删除缓存，并不是更新缓存？**

更新：一分钟更新了二十次数据，就要更新二十次缓存(有时缓存内容需要经过计算)，但是用户只请求了一次，那二十次的计算实际上是没必要的

删除：等到用户下次访问的时间进行一次计算

如果缓存更新成功，但是数据库更新失败，会导致数据不一致

**如何保证Redis 和 Mysql 的数据一致性**

数据不一致的情况：

1.先删Redis缓存，还未更新MySQL，另一个线程来读，缓存为空，去mysql读到更新前的旧数据

2.先更新MySQL，再删Redis缓存，还未删除缓存前，去缓存中读到缓存中的旧数据

解决方案

延迟双删策略，删除redis缓存，再更新mysql，延迟休眠500ms,再次删除redis缓存(延迟时间必须要大于更新redis的时间)

热点数据延迟双删后，大量请求到数据库，造成击穿

请求时，检查缓存是否存在，不存在再上锁去数据库找，其他等待锁的线程枷锁失败等待200毫秒继续从缓存中读取

2.异步更新缓存

mysql数据修改后，把binlog推送(消息队列)给redis,根据binlog更新缓存

**redis中基本数据结构**

键类型：字符串

值类型：字符串（简单动态字符串），

列表（双向链表-数据量大和压缩链表-数据量小），

哈希表（压缩链表-数据小时和哈希表），

集合（整数数组-大量整数和哈希表），

有序集合(跳表)

**动态字符串**

字符串长度可以调整，不用重新分配内存(长度，未使用空间，字符数组)

**压缩链表**

多个元素存在一个连续内存中(前缀长度，数据，后缀长度)

**整数数组**

所有整数有序存储在一个连续的内存数组中

**redis为什么快，为什么性能高**

数据存在内存中，读写快

内部使用高效的数据结构，数据操作比较简单

单线程,避免上下文切换

多路I/O复用模型，非阻塞I/O

使用多线程处理网络IO

**Redis分布式锁**

setnx:键不存在则插入键值对，返回1；存在则不执行任何操作，返回0

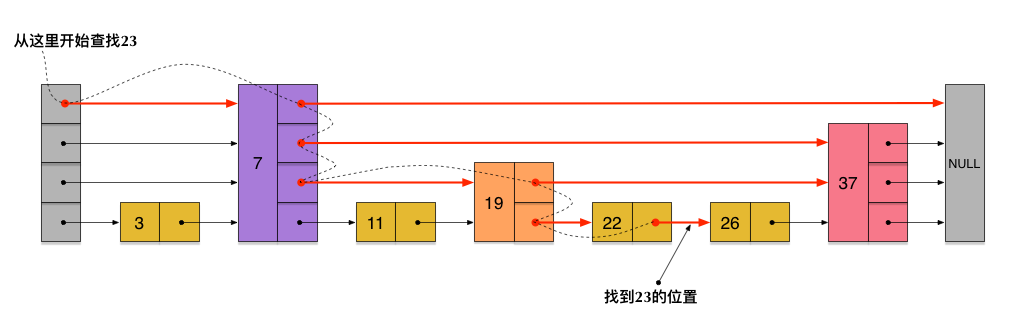
先拿setnx来争抢锁，抢到之后，再用expire给锁加一个过期时间防止锁忘记了释放。

如果在setnx之后，执行expire之前进程意外crash或者要重启维护了，这个锁就永远得不到释放了，使用set指令把setnx和expire合成一条指令来用

**redis中zset底层数据结构**

跳表(skipList)，查询，插入，删除 复杂度都为O(lgn)的数据结构，和平衡树复杂度一致

优点：内存更小，更新节点比B树更加节约内存，时间复杂度和平衡树一致，跳表容易实现，不用旋转节点， zset还有范围查找功能，跳表查找的更快



是否能用跳表做数据库索引？

1.B+树的节点可以存储很多关键字，而且单个节点大小可以与磁盘页对其（默认16KB），一次IO就能传输一整个节点（跳表可能会出现跨页IO）,大幅减少磁盘IO次数

2.跳表的索引层建立具有随机性，磁盘不能对链表进行预读，会产生大量随机IO（磁盘预读：从本次被预读的位置开始，顺序向后读取一定长度的数据放入内存）

3.B+树检索效率比跳表高

**Redis 持久化**

**RDB:**默认，创建快照保存某个时间点的所有数据。体积小，恢复速度快，不能实时，会丢失部分数据

**AOF:**记录增删改的指令，追加到AOF文件中。启动时执行AOF文件中的命令，还原数据库状态。实时，但是体积大，速度慢。AOF重写机制：文件过大时，set方式重新记录在新的文件中并替换，减少AOF中的冗余命令

**混合持久化：**快照+之后的增删改命令，结合RDB加载快、文件小及AOF实时的特点

**redis内存不足时的缓存删除策略**

返回错误，不删除任何键

删除最久没用的键

随机删除键

删除使用频率最少的键

删除将过期的键

**Redis如何判断key是否过期**

Redis中有expire字典存放每个key的过期时间

**redis缓存和本地缓存(HashMap)的优缺点和使用场景，分别存放什么数据？**

本地缓存：适合在单服务器上使用

优：访问速度快

缺：无法大容量存储，无法集群部署，无法持久化

redis缓存：支持持久化和分布式部署

优：支持大容量存储，保证数据一致性，读写分离高可用

缺: 在不同机器间通过网络来操作，性能略低于本地缓存

**判断key是否为热点key**

用户消费的数据远大于生产的数据(热点商品，新闻，评论等)，读多写少

请求分片集中，超过单个server的性能极限

服务端读数据进行访问时，会对数据进行分片切分，对某一主机 Server 上相应的 Key 进行访问，当访问超过 Server 极限时，就会导致热点 Key

**redis对应的三方框架**

jedis和redisson(推荐使用)

**Redis6.0之前是单线程还是多线程**

6.0之前都是单线程,内部读写都是单线程的，

多线程存在执行顺序不确定的问题，带来线程切换、加锁等性能问题

**Redis 6.0为什么要引入多线程**

单线程能处理100,000 QPS,随着业务扩展，需要更大的QPS

Redis的网络读写占大部分CPU时间，多线程通过IO多路复用，处理网络IO

**SETNX 实现分布式锁有哪些问题**

存在单点风险，如果存储的分布式锁key挂掉的话(突然宕机)，存在丢锁风险，会造成多个客户端同时握有锁，导致分布式锁失败。

**Redis集群搭建**

至少3台服务器，安装redis,并启动

redis-cli --cluster create .. --cluster-replicas 1创建集群,表示每个主节点需要1个从节点

redis-cli --cluster check 检查集群

redis-cli --cluster info 查看集群

redis-cli --cluster fix 修复集群

redis-cli --cluster add-node 扩容

redis-cli --cluster del-node 缩容

redis-cli --cluster call 集群中执行命令

redis-cli --cluster rehard 迁移槽位

redis-cli --cluster rebalance 平衡集群

**Redis 哨兵模式配置**

配置主节点，添加requirepass参数

配置从节点，添加slaveof参数，指定主节点ip和端口

redis-sentinel：启动哨兵进程

#### **redis主从和集群可以保证数据一致性吗 ？**

redis 主从和集群在CAP理论都属于AP模型，即在面临网络分区时选择保证可用性和分区容忍性，而牺牲了强一致性。这意味着在网络分区的情况下，Redis主从复制和集群可以继续提供服务并保持可用，但可能会出现部分节点之间的数据不一致。

**Redisson分布式锁的实现**

Set k v nx(不存在则插入) px(过期时间未毫秒)

增加了看门狗机制，快到过期时间但是任务没执行完会自动延长过期时间

获取锁时生成唯一的uuid，确保只有锁的持有者才能释放锁

一致性哈希算法

最终一致性

# Es数据库

**es近实时机制**

数据先写入缓冲区，每隔一秒将数据刷新一次，被索引引用到

使用场景

全文搜索引擎：模糊搜索、多字段搜索、分词搜索等

日志分析：ELK 是日志分析领域的经典解决方案。它可以高效地存储、索引和分析海量的日志数据。