Redis

> 缓存三兄弟 击穿 雪崩 穿透

击穿：缓存过期

1. 设置逻辑不过期（异步更新缓存过期时间 其他线程先返回过期的数据 只有一个线程更新缓存（锁）） 高可用
2. 强一致 互斥锁 直到更新完毕

雪崩：大量KEY失效，随机设置过期时间 或者 热点数据分布存储 热

数据永不过期 多级缓存 限流策略。。。

穿透：访问不存在的key 缓存没有 sql返回NUll后也不会存在redis 大量访问DB 导致DB负荷大

1.返回null值 并存在缓存里 设置一个过期时间防止负荷大

2.布隆过滤器 先查有没有 hash求值存在 01位图里面 因为是hash所以有误判率 可以设置为5%左右 redisson和guava有实现

> 双写一致

延迟双删 和读写锁 异步（canal和MQ通知）

> 持久化 *RDB和AOP*

一个存快照 ---> COPY ON WRITE

一个存写命令（这个更好 有三种设置 always everysecond和系统自己决定）

> 数据过期策略

因为不能一直存着 空间有限和性能有限

惰性删除 设置TTL 用到的时候会检查是否过期 检查到过期了再删除 对CPU友好但对内存不友好 因为没及时删

定期删除，定期对KEY检查（一定数量的KEY） 但是难以确定时长和频率 难以优化

SLOW模式 定时任务 默认10hz 每次不超过25ms (为了不影响主进程)

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

两种策略配合用最好

> 数据淘汰策略

redis内存不够 要删除了 怎么办？ 淘汰策略

八种策略 volatile:对设置了TTL的key来操作 allkeys 对所有keys操作

random lru lfu 三种策略 3*2 还有前两种策略

1. noeviction:默认策略 满了不写新数据
2. volatile-ttl: 对设置TTL的key，比较key的剩余TTL值，TTL越小越先淘汰
3. allkeys-random: 对全体key，随机进行淘汰
4. volatile-random:对设置了TTL的key 随机进行淘汰
5. allkeys-lru:对全体key,基于LRU算法进行淘汰。
6. volatile-lru:对设置ttl的key，基于LRU算法进行淘汰。
7. allkeys-lfu:对全体key，基于LFU算法进行淘汰。
8. volatile-lfu:对设置了TTL的key 基于LFU算法进行淘汰

有冷热区分：allkeys-lru LFU不准确

没有冷热区分：allkeys-random

有置顶需求：volatile-lru （置顶数据不过期）

短时高频：lfu策略

allkeys-lru淘汰数据 留下来的都是经常访问的热点数据

默认是**noeviction** 满了会保存

> 分布式锁

不同数据在不同的服务器 集群部署 锁的服务器 不同服务器不同线程共享一把锁

setnx命令 要加失效时间 防止死锁

可以使用 不同的锁 **Key** 来锁定不同的业务，确保多个业务逻辑之间互不干扰。

锁的失效时间可能会比业务执行时间短 怎么办？

给锁续期

redisson实现的分布式锁 已经实现了给锁续期

看门狗机制

释放锁 同时要通知看门狗 不需要再监听了

其他线程 while循环尝试获取锁 但是会设置循环次数限制（获取时间限制） 不会一直循环导致死循环

基于lua脚本

redisson锁也是可重入的 用hash结构记录线程id key id 和重入次数 重入次数变0就删除信息

主从一致性

RedLock红锁 多个redis实例上创建锁 n/2+1 防止主节点宕机

但是实现复杂性能差

非要实现强一致性 应该使用zookeeper实现的分布式锁 用临时节点来管理锁 搞完就自动删除节点（其他线程监听） 让当前最小的节点来获取锁 （排队）

> 其他面试问题

集群方案

主从复制 哨兵模式 (Sentinel) 分片Sharding模式 Cluster模式

> 主从复制

读写分离 写在主节点 同步到从节点 来读 故障只能手动升级从节点

实际上

还有

用replid id判断是否一致 来判断是否第一次同步 因为服务器变了的话 就要重新从0复制（全量同步）

执行bgsave用RDB文件来同步 RDB期间的命令（部分的AOP）(通过 Replication Buffer) 发到从节点来执行 防止漏了

offset判断是否落后 要不要更新 如果 `repl_offset` 发生偏移，则进行 增量同步。

> 哨兵模式

监控节点是否正常工作

master故障 会将一个slave升级

发生了故障转移 也要通知到客户端 写操作转移到新的master

基于心跳机制来监控 每隔1秒发Ping命令

主观下线和客观下线

前者是 没响应就主观下线 后者是超过指定数量的哨兵都发现下线了 那就客观下线

> 哨兵选择顺序

判断主和从节点断开时间长短 超过指定值就排除该从节点-》priority值（越小优先越高）---》**offset**值（越大数据越多 优先级越高）**---**》运行ID（越小越高）

> 脑裂问题

网络原因 主节点没寄 哨兵以为寄了 去选一个新的主节点 但是写入数据还在旧的主节点在写 网络恢复后 旧主节点被强制降为从节点 数据被清空

设置最少的slave节点 有slave才能写

设置同步延迟不超过设置时间

达不到要求就拒绝请求

1主1从就够了

> *Sharding*分片模式

应用程序或代理层决定某个 **key** 应该存储在哪个 **Redis** 实例 使用 `hash(key) % N` 计算 key 应该存储在哪个 Redis 实例上。 可以主从 也可以不主从

手动！

> *Sharding*模式故障怎么办？

手动！ 如果使用了主从架构，可以借助 **Sentinel** 进行自动主从切换，但 **Sentinel** 仅管理单个实例，不会自动迁移数据。

Cluster模式

Redis 自动管理分片

多个master 每个master保存不同数据 多个slave

master之间通过ping监测（心跳）彼此健康状态

访问任意节点 会转发到正确节点 --->哈希槽 CRC16 %16384 （像一致性哈希 但是一致性哈希有虚拟节点 负载均衡）

> 如何分片的？

每个 Redis 节点会负责一部分哈希槽，系统会在集群初始化时确定每个节点负责的哈希槽范围。每个哈希槽都有明确的归属节点。

Redis Cluster 采用 哈希槽（**Hash Slot**） 机制进行数据分片。

如果请求的哈希槽对应的节点不存在或无法访问，Redis Cluster 会返回 `MOVED` 错误，并重定向到正确的节点。即使某个节点失效，集群的路由机制会确保哈希槽总是映射到一个存在且可访问的节点上。

每个节点有自己的分配槽范围

Redis Cluster的节点之间会共享消息，每个节点都会知道是哪个节点负责哪个范围内的数据槽

> *ask*异常和*moved*异常的区别？

`MOVED` 错误表示请求的哈希槽已经被迁移到集群中的其他节点。客户端向错误的节点发送了请求，但该节点知道数据所在的正确节点，并会返回正确的目标节点信息。客户端收到这个错误后，应该重新向指定的节点发送请求。

集群扩容或节点迁移：当集群中的某个节点负责的哈希槽被迁移到其他节点时，原节点会返回 `MOVED` 错误，提示客户端向正确的节点重新发送请求。

节点故障转移：如果某个节点故障并且它负责的哈希槽被迁移到其他节点，Redis 会通过 `MOVED` 错误通知客户端。

`ASK` 错误表示请求的哈希槽正在进行 迁移中，但目标节点目前是该槽的 临时接管节点。此时，客户端应该立即向目标节点请求数据，但注意这是一种临时状态，最终哈希槽会被迁移到新节点。

> 临时接管节点

当集群发生扩容、节点故障修复或者手动迁移哈希槽时，某些哈希槽的数据可能需要从一个节点迁移到另一个节点。这种迁移不会立即完成，而是需要一个过渡阶段。在这个阶段中，某些节点会暂时接管该哈希槽的请求处理，直到迁移完成。

选一个节点作为临时接管节点

如果 节点 **C** 已经接管了该哈希槽的数据（例如数据已经完全迁移），节点 **C** 会直接返回该哈希槽的数据。

如果 节点 **C** 没有接管数据（因为数据还在迁移中），那么 节点 **C** 会遇到以下几种情况：

- 节点 **C** 可以直接访问原始节点 节点 **B** 请求数据，或者
- 节点 **C** 可以返回给客户端一个“缺失数据”的信息，提示客户端稍后重试。

> *16384?*

16384 是 **2^14**，是一个二的幂次数。使用二的幂次作为哈希槽的数量通常有助于优化哈希操作，

> 怎么扩容？过程？期间能服务吗？

使用 `CLUSTER MEET` 命令将新节点加入集群。这个命令会将新节点的 IP 和端口添加到集群的成员列表中。

使用 `CLUSTER ADDSLOTS` 命令将一部分哈希槽分配给新节点。集群会从现有节点迁移一部分哈希槽到新节点。Redis 提供了 `reshard` 工具来自动化这一过程。

这个过程是逐步迁移的，确保在节点扩容期间不会一次性迁移大量哈希槽，以避免对性能造成过大影响。

数据会根据哈希槽的重新分配从源节点迁移到目标节点。这是通过 **MIGRATE** 命令进行的，数据会逐步地迁移，保证每次迁移的数据量不会对性能产生显著影响。

扩容时，哈希槽的迁移是逐步进行的，并且每次迁移的哈希槽不会过多，因此集群能继续服务。Redis 会在迁移期间保持数据的可用性，客户端会得到重定向指令，而不是完全不可用。

其次是读写分离，读操作读旧节点，写操作重定向 （类似于缓存击穿问题 设置逻辑不过期）

> 节点间怎么通信？

TCP协议

gossip通信： 定期Ping检查状态和可用性 更新自己的节点列表

命令通信

> *Cluster*模式故障了怎么办？

跟主从不一样

集群之间 会ping命令

主观下线

1.节点1定期发送ping消息给节点2

2.如果发送成功，代表节点2正常运行，节点2会响应PONG消息给节点1，节点1更新与节点2的最后通信时间

3.如果发送失败，则节点1与节点2之间的通信异常判断连接，在下一个定时任务周期时，仍然会与节点2发送ping消息

4.如果节点1发现与节点2最后通信时间超过node-timeout，则把节点2标识为pfail状态

客观下线：

如果接收到的ping消息中包含了其他pfail节点，这个节点会将主观下线的消息内容添加到自身的故障列表中，故障列表中包含了当前节点接收到的每一个节点对其他节点的状态信息

当前节点把主观下线的消息内容添加到自身的故障列表之后， 对其他节点发消息 说这个节点寄了，其他节点收到后也尝试ping 也发现寄了 那就主管下线

至少要有超过 **N/2** 的节点都认为某个节点（比如 节点 **B**）故障，才能确认该节点是 客观下线。

集群内部每个节点会定期或按需计算 主观下线（**pfail**） 的节点数量。这个计算主要是检查：

> 故障恢复？

迁移数据

提升从节点为主节点：当一个 **master** 节点 被标记为 客观下线 后，集群会自动选择一个该节点的从节点 来提升为新的 **master** 节点。此时，从节点会接管原来主节点负责的哈希槽。

迁移哈希槽：集群会将故障节点（原 master 节点）负责的哈希槽迁移到新的 master 节点或其他正常的节点上。

其实也没啥 和主从是一套的

> *smart*客户端

自动化 性能！

Smart 客户端（或称为 智能客户端）是指在客户端和服务器之间增加了智能化处理的客户端工具或库。它不仅仅是单纯地向服务器发送请求并接收响应，还具备一些智能功能，例如负载均衡、故障转移、数据分区处理、自动重试等。

一些底层

> *Redis*是单线程的

线程安全问题 避免不必要的上下文切换 使用非阻塞IO （NIO）

NIO主要就是实现了高效的网络请求

内核空间和用户空间

NIO的一种 **IO**多路复用

> *BIO AIO NIO*

BIO是阻塞IO 一直阻塞直到完成拿到数据

NIO不会阻塞 但是要等 反复read 此时非阻塞 （忙等） 数据就绪后 用户进程阻塞 拿数据

AIO是改进版的NIO 和NIO区别是 NIO程序决定channel准备好的操作 AIO交给内核主动来搞（异步）

但是AIO没有好的实现

比喻： BIO是充电桩充电 一直阻塞

这部分看网络相关