链滴

Redis 必知必会

作者：Hefery

原文链接：https://ld246.com/article/1641183779393

来源网站：链滴

Redis

Redis简介

Redis：高性能（运行速度很快，并发很强，跑在内存上）的 NoSql(Not Only SQL) 系列的基于键值非关系型数据库

特点

- 速度快，高性能：10WOPS，每秒可实现10W次读写操作（数据存储在内存；C语言精简编写；单程）
- 支持持久化：Redis所有数据保持在内存中，对数据的更新将异步地保存到磁盘上
- 支持多种数据结构：String；Hash；Linked List；Set；Sorted Set
- 支持多种语言：Java；PHP；Python；Ruby；Lua；Nodejs
- 功能丰富：发布订阅；Lua脚本；事务；pipeline

Redis快的原因

- 采用了多路复用io阻塞机制
- 数据结构简单，操作节省时间
- 运行在内存中，自然速度快

Redis支持的键值数据类型：

- 字符串类型 String
- 哈希类型 Hash
- 列表类型 List
- 集合类型 Set
- 有序集合类型 SortedSet

应用场景

- 缓存（数据查询、短连接、新闻内容、商品内容等等）
- 聊天室的在线好友列表
- 任务队列（秒杀、抢购、12306等等）
- 应用排行榜
- 网站访问统计
- 数据过期处理（可以精确到毫秒）
- 分布式集群架构中的Session分离

主流的NOSQL产品

- 键值(Key-Value)存储数据库

相关产品： Redis

典型应用： 内容缓存，主要用于处理大量数据的高访问负载。

数据模型： 一系列键值对

优势： 快速查询

劣势： 存储的数据缺少结构化

- 列存储数据库

相关产品：HBase

典型应用：分布式的文件系统

数据模型：以列簇式存储，将同一列数据存在一起

优势：查找速度快，可扩展性强，更容易进行分布式扩展

劣势：功能相对局限

- 文档型数据库

相关产品：MongoDB

典型应用：Web应用（与Key-Value类似，Value是结构化的）

数据模型： 一系列键值对

优势：数据结构要求不严格

劣势： 查询性能不高，而且缺乏统一的查询语法

- 图形(Graph)数据库

相关数据库：Neo4J、InfoGrid、Infinite Graph

典型应用：社交网络

数据模型：图结构

优势：利用图结构相关算法

劣势：需要对整个图做计算才能得出结果，不容易做分布式的集群方案

Redis安装

Windows10

```
官网：https://redis.io/
    中文：http://www.redis.net.cn/
    解压：
        redis.windows.conf：配置文件
        redis-cli.exe      ：客户端
        redis-server.exe  ：服务器端
```

CentOS7

```
安装：
    tar -xzf redis-3.0.7.tar.gz
    ln -s redis-3.0.7 redis
    cd redis
    make && make install
```

```
redis-server启动：
```

最简启动：redis-server (默认配置)
动态参数：redis-server --port 6380
配置文件：redis-server configPath
在redis目录下创建config目录，并把redis.conf复制到config目录下，更名为redis-6381.conf
去掉conf文件的注释：cat redis-6381.conf | grep -v "#" | grep -v "^$" > redis-6381-hefery.conf
编辑redis-6381-hefery.conf
启动：在redis路径下
redis-server redis-6381-hefery.conf
redis-cli客户端：
连接：redis-cli -h 127.0.0.1 -p 7000 -a 123456
退出：quit
关闭：
停止redis-server服务: redis-server &
redis-cli -p 6379 shutdown
查看启动：
ps -ef | grep redis
ps aux | grep redis
Windows连接Linux Redis：
关闭防火墙：systemctl stop firewalld
redis.conf设置bind 0.0.0.0

Redis API

Redis的数据结构：Redis 存储的是 key:value 格式的数据，其中key都是字符串，value有不同的数结构

字符串类型 string

哈希类型 hash ： map格式

列表类型 list ： linkedlist格式。支持重复元素

集合类型 set ： 不允许重复元素

有序集合类型 sortedset：不允许重复元素，且元素有顺序

字符类型 string

结构：key : value

场景：
缓存：热点数据、对象缓存、全页缓存
计数器：个人主页的访问量（incr userid:pageview）
分布式锁：setnx只有不存在才会添加成功
数据共享：spring-session-data-redis

API：
设置；
不管key是否存在：set key value
key不存在才设置：setnx key value
key是存在才设置；set key value xx
批量设置key： mset key1 key2...
设置新值返回旧值：getset key newvalue
value追加： append key value
设置字符串指定下标的所有值：setrange key index value

获取：
单个获取：get key
批量获取：mget key1 key2...
字符串长度：strlen key（一个中文两个字节）
字符串指定下标的所有值：getrange key stsrt end（index从0开始）
删除：del key
自增：
整数自增：incr key（自增1）、incrby key 66（自增66）
浮点自增：incrbyfloat key 6.6（自增6.6）
自减：
整数自减：decr key（自减1）、decrby key 66（自减66）
浮点自减：decrbyfloat key 6.6（自减6.6）

哈希类型 hash

结构：key : field(n)-value(n)

场景：
计数器：个人主页的访问量（hincrby user:1:info pageview count）(incr)
全局ID：分库分表(incrby)
购物车：
点赞、签到、打卡：
微博ID=t1001、用户ID=u3001、点赞用户：like:t1001来维护所有为这条微博点赞的用户
微博点赞：sadd like:t1001 u3001
取消点赞：srem like:t1001 u3001
是否点赞：sismember like:t1001 u3001
点赞用户：smember like:t1001 u3001
点赞数量：scard like:t1001
用户关注：follow 关注 fans 粉丝
相互关注：
sadd 1:follow 2
sadd 2:fans 1
sadd 1:fans 2
sadd 2:follow 1
我关注的人也关注她：交集
sinter 1:follow 2:fans
可能认识的人：
用户1可能认识的人(差集)：sdiff 2:follow 1:follow
用户2可能认识的人(差集)：sdiff 1:follow 2:follow
排行榜：新闻ID=6001
新闻点击数+1：zincrby hotNews:20211111 1 n6001
今天点击最多：zrevrange hotNews:20211111 0 1 withscores

API：
设置：
设置hash key对应field的value：hsetnx key field value（如果field已存在则失败）
设置hash key filed的value：hset key field value
批量设置hash key filed的value：hmset key1 key2...
获取：
获取hash key filed的value：hget key field
批量hash key filed的value：hmget key1 key2...
获取所有hash key对应的key-filed的value：hgetall key
获取所有hashkey对应的field：hkeys key

获取所有hash key对应的filed的value：hvals key
获取hash key field数量：hlen key
删除：
删除hash key filed：hdel key field
自增：
整数自增：hincr key field（自增1）、hincr key field 66（自增66）
浮点自增：hincrbyfloat key field 6.6（自增6.6）
自减：
整数自减：hdecrby key field（自减1）、hdecrby key field 66（自减66）
浮点自减：hdecrbyfloat key field 6.6（自减6.6）
判断：
判断hash key是否存在field：hexists key field

列表类型 list

结构：key : elements(有序；可重)

场景：
用户消息时间线timeline：双向链表，插入有序
消息队列：blpop、brpop，可设置超时时间(队列：rpush blpop左头右尾，右进左出 栈：rpush rpop先进后出)

API：
设置：
从右向list插入值：rpush key value1 value2...
从左向list插入值：lpush key value1 value2...
在list指定的value值前或后插入newvalue：linsert key before|after value newvalue
设置list指定index的item为newvalue：lset key index newvalue
弹出：
从list左侧弹出item：lpop key
从list右侧弹出item：rpop key
根据count的值弹出所有value相等的item：lrem key count value
count > 0：从左到右，弹出最多count个value相等的item
count < 0：从右到左，弹出最多Mash.abs(count)个value相等的item
count = 0：删除所有value相等的item
按照index范围修剪list：itrim key start end
lpop阻塞版本：blpop key timeout（timeout为超时时间，为0即永不阻塞）
rpop阻塞版本：brpop key timeout（timeout为超时时间，为0即永不阻塞）
获取：
获取list的(start, end]的所有item：lrange key start end
PS：a b c d e f：index从左到右0~5，从右到左-1~-6
获取list指定index的item：lindex key index
获取list长度：llen key

集合类型 set

结构：key : values（无序；无重）

API：
添加：
添加value：sadd key value（如果value已存在则失败）
删除：
删除value：srem key value

从set随机弹出一个元素：spop key
获取：
获取set大小：scard key
从set随机挑选count个元素：srandmember key count
获取set所有value：smembers key
判断；
判断value是否在set中：sismember key（1存在）
集合间的操作：
差集：sdiff key1 key2
交集：sinter key1 key2（共同关注）
并集：sunion key1 key2

有序集合 sortedset

结构：key : [score-value]（有序；无重）

场景：
排行榜

API：
添加：
添加score-value：zadd key score value（score可重，value不重）
删除：
删除score-value：zrem key value
删除指定排名的升序元素：zremrangebtrank key stsrt end
删除指定分数范围内的升序元素：zremrangebyscore key minscore maxscore
获取：
获取value的score：zscore key value
增加或减少value的score：zincrby key increscore value
获取集合元素个数；zcard key
获取指定范围(按score升序)内元素：zrange key start end
获取指定分数范围内的升序元素：zrangebyscore key minscore maxscore
zrevrank
zrevrange
zrevrangebyscore

通用命令

keys：
所有的key：keys *
key得数量：dbsize
key的正则：keys [pattern]

exists：
检查key是否存在：exists key

del：
删除指定的key-value：del key1 key2...

expire：
设置key的过期时间：expire key seconds
查看key的剩余时间：ttl key（-2代表key已经不存在）
去掉key的过期时间：persist key（（-1代表key存在，并且没有过期时间）

type：
查看key的类型：type key（string、hash、list、set、zset、none）

Redis客户端

Jedis

Redis的java客户端

Jedis配置参数	参数含义	默认
资源数控制		
maxTotal	资源池最大连接数	8
maxIdle	资源池允许最大空闲连接数	8
minIdle	资源池允许最小空闲连接数	0
jmxEnabled	是否开启jmx监控	true
借还参数		
blockWhenExhausted	当资源池用尽后，调用者是否要等待。当为true，maxWaitMillis才生效	true
maxWaitMillis	当资源池连接用尽后，调用者的最大等待时间（单位为毫）	-1：永不超时
testOnBorrow	向资源池借用连接时是否做连接有效性检测（ping），无连接会被移除	false
testOnReturn	向资源池归还连接时是否做连接有效性检测（ping），无连接会被移除	false

建议
面讨论
议maxTotal
面讨论
议开启
建议使用默认值
不建议使用默认值
建议false
建议false

maxTotal：根据业务而定
命令平均执行时间0.1ms=0.001s；业务需要5w QPS；maxTotal理论值=0.001*5w=50，实际值要更大
业务希望Redis并发量；客户端执行命令时间；Redis资源(应用个数Nodes * maxTotal <= Redis大连接数maxclients)

maxIdle：建议maxIdle=maxTotal
减少创建新连接的开销
maxIdle：建议预热maxIdle
减少第一次启动后的新连接开销

获取超时：
原因：
慢查询阻塞，连接池被堵住
资源池参数配置不合理（QPS高，连接池容量小）
连接泄露（没有close，比较难定位）

Redis功能

慢查询

慢查询发生在Redis执行命令阶段

配置：
- slowlog-max-len：
  - 默认128（先进先出的队列 + 固定长度 + 保存在内存）
  - 不要设置过大，默认10ms，通常设置1ms
- slowlog-log-slower-than：默认1w
  - 1. 慢查询阈值（单位：微秒）
  - 2. slowing-log- slower-than=0，记录所有命令
  - 3. slowing-og- slower-than<0，不记录任何命令
  - sawlog-log- slower-than不要设置过小，通常设置1000左右
- 动态配置：
  - config set slowlog-max-len 1000
  - config set slowing-og- slower-than 1000

API：
- 获取慢查询队列：slowlog get [n]
- 获取慢查询队列长度：slowlog len
- 清空慢查询队列：slowlog reset

流水线Pipeline

Redis执行命令都是微妙级别
pipeline每次条数要控制(网络)

注意每次 pipeline携带数据量
pipeline每次只能作用在一个 Redis节点上

命令个命令)	**N个命令操作**	**1次pipeline(**
时间次网络传输时间+1次命令执行时间	**n次网络传输时间+n次命令执行时间**	
数据量	**1条命令**	**n条命令**

发布订阅

角色：
- 发布者publisher：
- 订阅者subscriber：
- 频道channel：

模型：
- 发布者(redis-cli-publisher) -> Sohutv频道(redis-server) -> 订阅者(redis-cli-subscriber)

API：
- 发布消息：publish channel message
- 订阅消息：subscribe [channel(n)]
- 取消订阅：unsubscribe [channel(n)]

BitMap

位图：二进制

API：
　给位图指定索引设置值：setbit key offset value
　获取位图指定范围(start到end，单位字节，如果不指定就获取全部)位值为1的个数：bitcount key start end]
　计算位图指定范围(start到end，单位字节，如果不指定就获取全部)第一个偏移量对应的值等于targtBit的位置：bitpos key targetBit [start] [end]

HperLogLog

基于HyperLogLog算法：极小空间完成独立数量统计

API：
　向hyperloglog添加元素：fadd key element1 element2...
　计算hyperloglog的独立总数：pfcount key1 key2...
　合并多个 hyperloglog：pfmerge destkey sourcekey1 sourcekey2...

GEO

GEO（地理信息定位）：存储经纬度，计算两地距离，范围计算等

API：
　增加地理位置信息：geo key longitude latitude member [longitude latitude member]
　获取地理位置信息：geopos key member [member]
　获取两个地理位置的距离；geodist key memberl member2 [unit] unit：m（米）、km（千米）、mi（英里）、ft（尺）
　获取指定位置范围内的地理位置信息集合：
　　georadius key longitude latitude radium|km|ft|mi
　　[withcoord]：返回结果中包含经纬度
　　[withdist] ：返回结果中包含距离中心节点位置
　　[withhash] ：返回结果中包含 geohash COUNT count：指定返回结果的数量
　　[COUNT count]：指定返回结果数量
　　[asc|desc] ：返回结果按照距离中心节点的距离做升序或者降序
　　[store key]：将返回结果的地理位置信息保存到指定键
　　[storedist key]：将返回结果距离中心节点的距离保存到指定键
　　georadiusbymember key member radium|km|ft|mi
　　[withcoord]：返回结果中包含经纬度
　　[withdist] ：返回结果中包含距离中心节点位置
　　[withhash] ：返回结果中包含 geohash COUNT count：指定返回结果的数量
　　[COUNT count]：指定返回结果数量
　　[asc|desc] ：返回结果按照距离中心节点的距离做升序或者降序
　　[store key]：将返回结果的地理位置信息保存到指定键
　　[storedist key]：将返回结果距离中心节点的距离保存到指定键

Redis持久化

Redis是一个内存数据库，当redis服务器重启，获取电脑重启，数据会丢失，可以将redis内存中的数持久化保存到硬盘的文件中

Redis持久化：Redis所有数据保存在内存中，对数据的更新异步保存到磁盘

持久化方式：RDB：拍快照、AOF：写日志

Redis持久化机制

1.RDB：默认方式，不需要进行配置
在一定的间隔时间中，检测key的变化情况，保存某个时间点的全量数据，然后持久化数据
1.编辑redis.windwos.conf文件
after 900 sec (15 min) if at least 1 key changed
save 900 1
after 300 sec (5 min) if at least 10 keys changed
save 300 10
after 60 sec if at least 10000 keys changed
save 60 10000
2.重新启动redis服务器，并指定配置文件名称
W:\JAVA\Redis\redis\windows-64\redis-2.8.9>redis-server.exe redis.windows.conf

2.AOF：日志记录的方式，可以记录每一条命令的操作。可以每一次命令操作后，持久化数据
1. 编辑redis.windwos.conf文件
appendonly no（关闭aof） --> appendonly yes （开启aof）
appendfsync always： 每一次操作都进行持久化
appendfsync everysec： 每隔一秒进行一次持久化
appendfsync no ： 不进行持久化

3.RDB-AOF混合：BGSave全量持久化，AOF增量持久化

API	RDB	AOF
启动优先级	低	高
体积	小	大
恢复速度	快	慢
数据安全	丢数据	策略决定
轻重	重	轻

RDB

触发机制：
save(同步)：通常会阻塞Redis
redis-cli执行save命令，redis-sever创建RDB二进制文件（新的文件替换旧文件）
bgsave(异步)：不会阻塞Redis，但会fork新进程
redis-cli执行bgsave命令，redis-sever执行fork()，创建子进程完成创建RDB二进制文件（新文件替换旧文件）
自动：
#save 900 1
#save 300 10
#save 60 10000
dbfilename dump-${port}.rdb # RDB文件名
dir /bigdiskpath
stop-writes-on-bgsave-error yes # 报错，终止bgsave
rdbcompression yes
rdbchecksum yes

复制方式：
全量复制：
debug reload：
shutdown：

RDB问题：
耗时-O(n)；耗性能-fork()消耗内存；Disk IO-IO性能
不可控，丢失数据

API	save	bgsave
IO类型	同步	异步
阻塞	阻塞	阻塞(fork阶段)
复杂度	O(n)	O(n)
优点	不会消耗额外内存	不阻塞户端命令
缺点	阻塞客户端命令	需要fork消耗内存

AOF

触发机制：
always：
redis-cli写命令刷新缓冲区，缓冲区中每条命令fync到硬盘的AOF文件
everysec：
redis-cli写命令刷新缓冲区，每秒把缓冲区fync到硬盘的AOF文件
no：
redis-cli写命令刷新缓冲区，OS决定缓冲区中命令fync到硬盘的AOF文件

AOF重写:
重写：
set hello world; set hello java; -> set hello java
incr counter; incr counter; -> set counter 2
作用：减少硬盘占用量;加速恢复速度
操作：
redis-cli执行bgrewriteaof，redis-sever执行fork()，创建子进程实现AOF重写
配置：

```
auto-aof-rewrite-min-size    # AOF文件重写需要的尺寸
auto-aof-rewrite-percentage  # AOF文件增长率
aof_current_size             # AOF当前尺寸(单位：字节)
aof-base-size                # AOF上次启动和重写的尺寸(单位：字节)
---------------------------------------------------------------------
appendonly yes
appendfilename  “appendonly-${port}.aof”
appendfsync everysec
dir /bigdiskpath
no-appendfsync-on-rewrite yes
auto-aof-rewrite-percentage 100
auto-aof-rewrite-min-size 64mb
```

自动：同时满足
aof_current_size > auto-aof-rewrite-min-size

(aof_current_size - aof-base-size) | (aof-base-size > auto-aof-rewrite-percentage)

API	always	everysec	no
优点	不丢失数据	每秒一次fsync，丢1s数据	用管
缺点	IO开销大	丢1s数据	可控

开发运维问题

fork操作

特点：
1. 同步操作：比较快
2. 与内存量息息相关：内存越大，耗时越长(与机型相关)
3. 查看fork info：latest_fork_usec

改善：
优先使用物理机或高效支持fork操作的虚拟化技术
控制Redis实例最大可用内存Lmaxmemory
合理配置Linux内存分配策略：vm.overcommit_memory=1
降低fork频率：放宽AOF重写自动触发时机，不必要全量复制

子进程开销和优化

1.CPU
开销：RDB和AOF文件生成，属于CPU密集型
优化：不做CPU绑定，不和CPU密集型应用部署

2.内存
开销：fork内存开销，copy-on-write会开一个副本
优化：echo never > /sys/kernel/mm/transparent_hugepage/enabled

3.硬盘
开销：AOF和RDB文件写入，iostat、iotop分析
优化：
不要和高硬盘负载服务(存储服务、消息队列)部署在一起
no-appendfsync-on-write=yes
根据写入量决定磁盘类型：ssd
单机多实例持久化文件目录考虑分盘

AOF追加阻塞

流程：
主线程写入到AOF缓冲区，每秒同步到硬盘，对比上次fsync时间(大于2s-阻塞；小于2s-通过)
定位：
Redis日志：Asynchronous AOF fsync is taking too long(disk is busy?)
命令info persistence：aof_delayed_fsync:100

Redis主从复制

主从复制

单机问题：机器故障；容量瓶颈；QPS瓶颈

作用：数据副本；拓展读性能
特点：
    一个master可以有多个slave
    一个slave只能有一个master
    数据流单向：master->slave

实现：
    salveoof：无需重启
        复制：redis-cli执行salveoof 127.0.0.1 6379命令，redis-cli-slave(6380)复制给redis-cli-mast
r(6379)
        取消：redis-cli执行salveoof no one命令，redis-cli-slave(6380)取消复制给redis-cli-master(
379)
    配置：统一配置，需要重启
        slaveof ip port
        slave-read-only yes

```
# redis-master-6379.conf
daemonize yes
pidfile /var/run/redis-master-6379.pid
port 6379
logfile "master-6379.log"
#save 900 1
#save 300 10
#save 60 10000
dbfilename dump-master-6379.rdb
dir /home/hefery/Downloads/Redis/redis/data/log
appendonly yes
appendfilename "appendonly-master-6379.aof"

# redis-slave-6380.conf
daemonize yes
pidfile /var/run/redis-slave-6380.pid
port 6380
logfile "slave-6380.log"
#save 900 1
#save 300 10
#save 60 10000
dbfilename dump-slave-6380.rdb=
dir /home/hefery/Downloads/Redis/redis/data/log
slaveof 127.0.0.1 6380
appendonly yes
appendfilename "appendonly-slave-6380.aof"


# 连接master-cli，查看master信息
> redis-server redis-master-6379.conf
> redis-cli
> info replication
# Replication
role:master
```

```
connected_slaves:0
master_repl_offset:0
repl_backlog_active:0
repl_backlog_size:1048576
repl_backlog_first_byte_offset:0
repl_backlog_histlen:0

# 连接slave-cli，查看slave信息
> redis-server redis-slave-6380.conf
> redis-cli -p 6380 info replication
# Replication
role:slave
master_host:127.0.0.1
master_port:6379
master_link_status:up
master_last_io_seconds_ago:10
master_sync_in_progress:0
slave_repl_offset:71
slave_priority:100
slave_read_only:1
connected_slaves:0
master_repl_offset:0
repl_backlog_active:0
repl_backlog_size:1048576
repl_backlog_first_byte_offset:0
repl_backlog_histlen:0

# 连接master-cli录入key，去slave-cli查看
> redis-cli
127.0.0.1:6379> keys *
(empty list or set)
127.0.0.1:6379> set hello world
OK
127.0.0.1:6379> exit
> redis-cli -p 6380
127.0.0.1:6380> get hello
"world"

# slave-cli不能写入key
127.0.0.1:6380> set hello hefery
(error) READONLY You can't write against a read only slave.
```

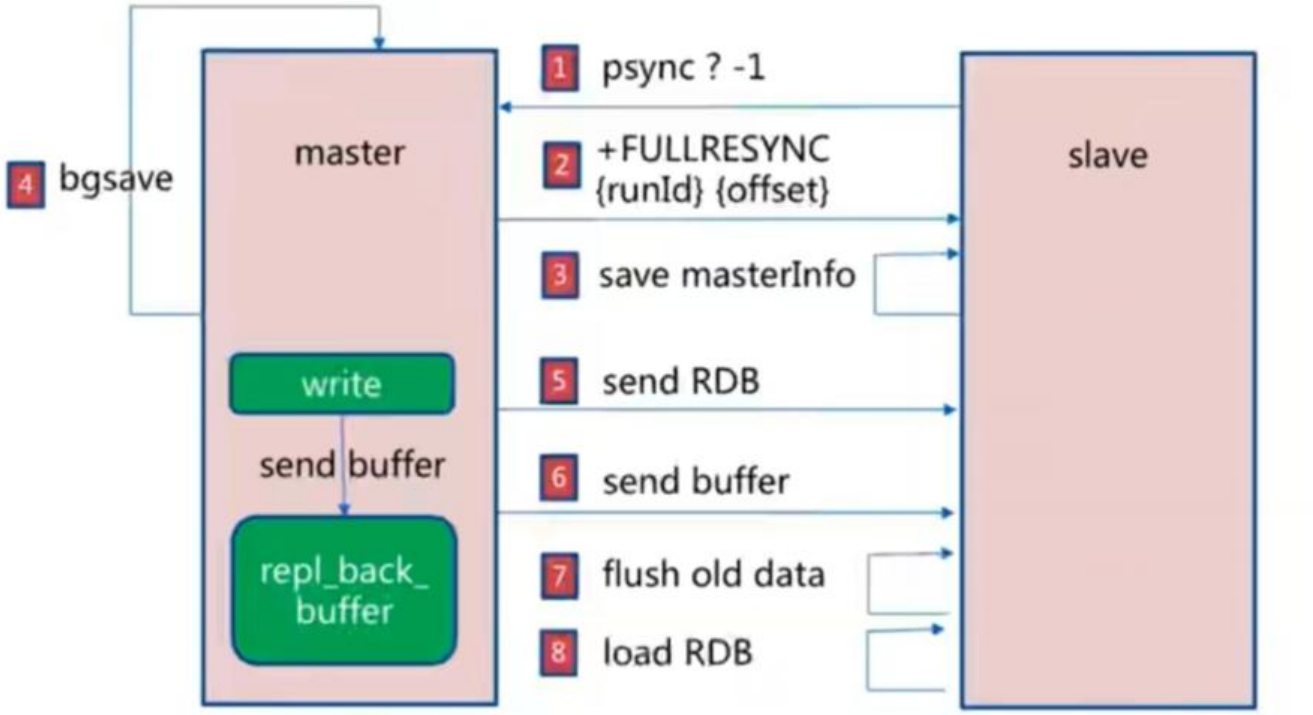
全量复制和部分复制

```
# 查看Redis的run_id
redis-cli -p 6379 info server | grep run
redis-cli -p 6380 info server | grep run
```

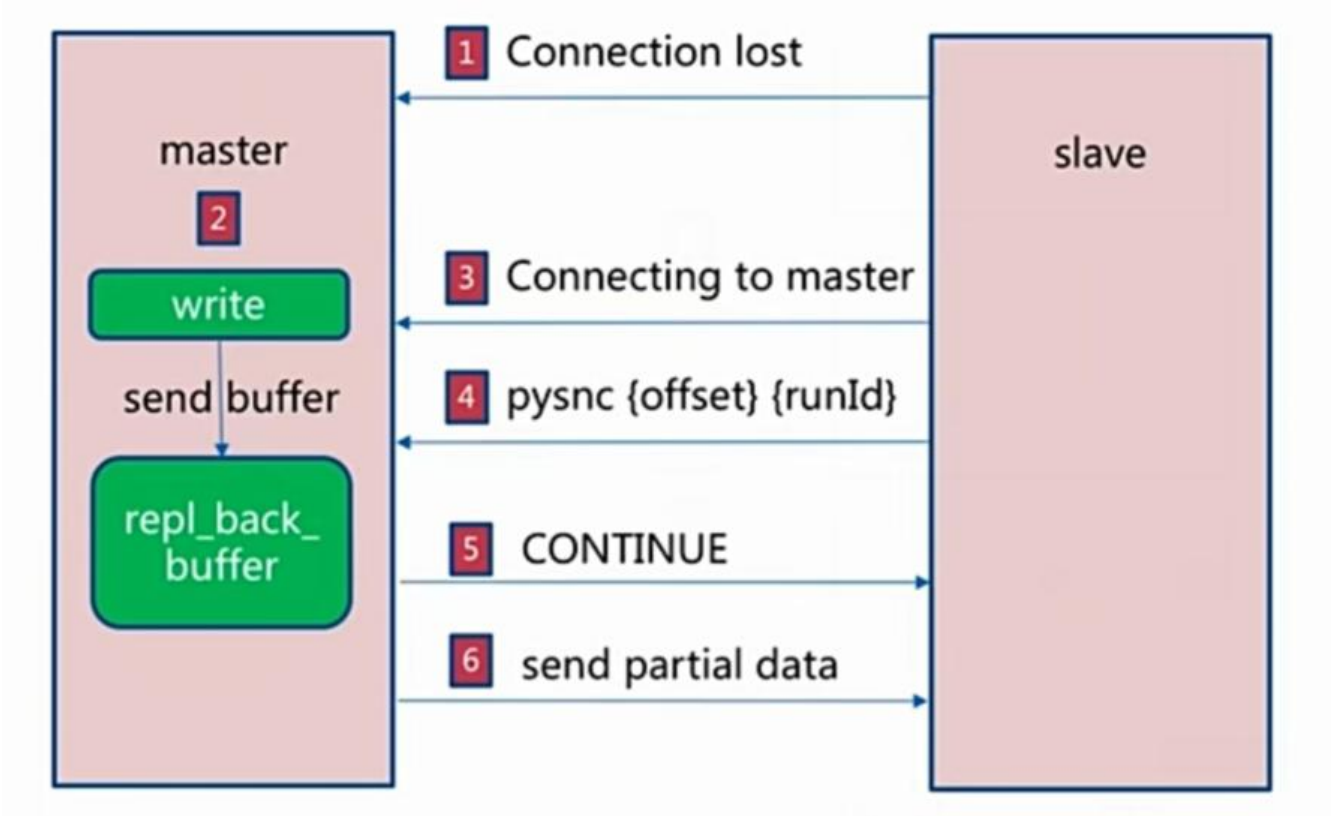
全量复制

```
全量复制开销；
    bgsave时间
    RDB文件网络传输时间
    从节点清空数据时间从
```

节点加载RDB的时间可能的AOF重写时间

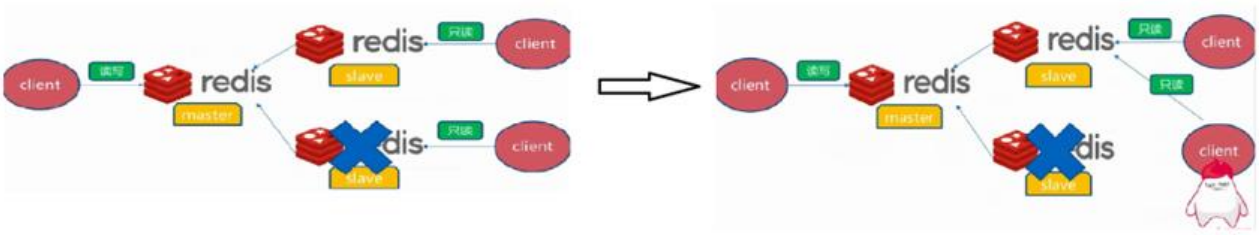


部分复制

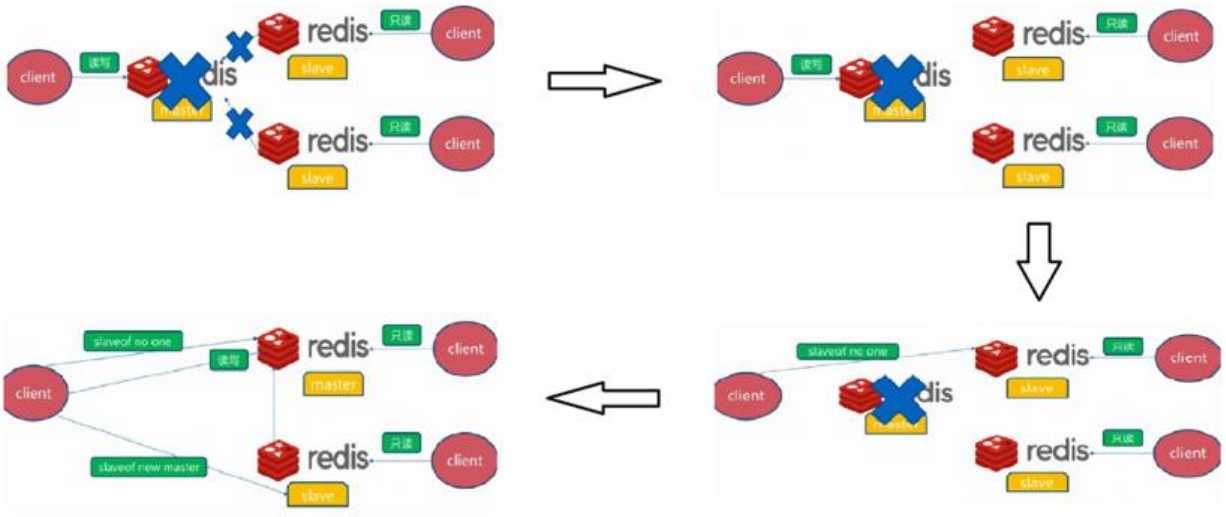


自动故障转移

slave宕机



master宕机



开发运维问题

读写分离

简述：redis-cli写数据在master，读数据在slave
问题：
　复制数据延迟
　读到过期数据
　从节点故障

主从配置不一致

1. maxmemory配置不一致，导致丢数据
2. 数据结构优化参数(hash-max-ziplist-entries)，导致内存不一致

规避全量复制

第一次全量复制无法避免
解决：
　夜间低峰处理

节点运行ID不匹配

解决：
    主节点重启(运行ID变化)
    故障转移(集群/哨兵)

复制积压缓冲区不足
原因：网络中断，部分复制无法满足
解决：增大复制缓冲区配置rel_backlog_size，网络增强

规避复制风暴

单主节点复制风暴：
    问题：主节点重启，多从节点复制
    解决：更换复制拓扑

单机器复制风暴：
    机器宕机后，大量全量复制
    主节点分散多机器

Redis Sentinel故障转移

搭建Redis Sentinel

```
# 配置主节点7000
vim redis-sentinel-7000.conf
port 7000
daemonize yes
pidfile /var/run/redis-sentinel-7000.pid
logfile "redis-sentinel-7000.log"
dir /home/hefery/Downloads/Redis/redis/data/log

# 配置从节点7001+7002
sed "s/7000/7001/g" redis-sentinel-7000.conf > redis-sentinel-7001.conf
sed "s/7000/7002/g" redis-sentinel-7000.conf > redis-sentinel-7002.conf

# 声明主从
echo "slaveof 127.0.0.1 7000" >> redis-sentinel-7001.conf
echo "slaveof 127.0.0.1 7000" >> redis-sentinel-7002.conf

# 启动redis-server
redis-server redis-sentinel-7000.conf
redis-server redis-sentinel-7001.conf
redis-server redis-sentinel-7002.conf

# 查看redis-server启动情况
ps -ef | grep redis-server
root      14251      1  0 22:52 ?        00:00:00 redis-server *:7000
root      16094      1  0 22:53 ?        00:00:00 redis-server *:7001
root      16499      1  0 22:53 ?        00:00:00 redis-server *:7002
root      16686 101512  0 22:53 pts/5    00:00:00 grep --color=auto redis-server

# 查看主节点分片情况
redis-cli -p 7000 info replication
```

```
# Replication
role:master
connected_slaves:2
slave0:ip=127.0.0.1,port=7001,state=online,offset=169,lag=0
slave1:ip=127.0.0.1,port=7002,state=online,offset=169,lag=1
master_repl_offset:169
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:2
repl_backlog_histlen:168


# 回到redis路径，Copy sentinel.conf到config路径下
cp sentinel.conf ./config/

# 精简sentinel.conf
cat sentinel.conf | grep -v "#" | grep -v "^$" > redis-sentinel-26379.conf

# 配置 redis-sentinel-26379.conf
port 26379
dir /home/hefery/Downloads/Redis/redis/data/log
daemonize yes
logfile "redis-sentinel-26379.log"
sentinel monitor mymaster 127.0.0.1 7000 2
sentinel down-after-milliseconds mymaster 30000
sentinel parallel-syncs mymaster 1
sentinel failover-timeout mymaster 180000

# 启动redis-sentinel
redis-sentinel redis-sentinel-26379.conf
# 查看redis-sentinel启动情况
ps -ef | grep redis-sentinel

# redis-cli连接redis-sentinel
redis-cli -p 26379

# 查看redis-sentinel-26379.conf发生改变
> cat redis-sentinel-26379.conf
port 26379
dir "/home/hefery/Downloads/Redis/redis-3.0.7/data/log"
daemonize yes
logfile "redis-sentinel-26379.log"
sentinel monitor mymaster 127.0.0.1 7000 2
sentinel config-epoch mymaster 0
sentinel leader-epoch mymaster 0
sentinel known-slave mymaster 127.0.0.1 7002
# Generated by CONFIG REWRITE
sentinel known-slave mymaster 127.0.0.1 7001
sentinel current-epoch 0

# 增加redis-sentinel-26380.conf+redis-sentinel-26381.conf
sed "s/26379/26380/g" redis-sentinel-26379.conf > redis-sentinel-26380.conf
sed "s/26379/26381/g" redis-sentinel-26379.conf > redis-sentinel-26381.conf
```

```
# 启动26380+26381
redis-sentinel redis-sentinel-26380.conf
redis-sentinel redis-sentinel-26381.conf
ps -ef | grep redis-sentinel

# 登录26379查看 slaves=2,sentinels=3
> info
# Server
redis_version:3.0.7
redis_git_sha1:00000000
redis_git_dirty:0
redis_build_id:b32e406b52b17276
redis_mode:sentinel
os:Linux 3.10.0-1160.42.2.el7.x86_64 x86_64
arch_bits:64
multiplexing_api:epoll
gcc_version:4.8.5
process_id:33120
run_id:c40b4fa3fb2f21de80e2fab27ce29176995c6c16
tcp_port:26379
uptime_in_seconds:39
uptime_in_days:0
hz:15
lru_clock:6662430
config_file:/home/hefery/Downloads/Redis/redis-3.0.7/config/redis-sentinel-26379.conf
# Sentinel
sentinel_masters:1
sentinel_tilt:0
sentinel_running_scripts:0
sentinel_scripts_queue_length:0
master0:name=mymaster,status=ok,address=127.0.0.1:7000,slaves=2,sentinels=3

ps -ef | grep redis
root     14251     1  0 22:52 ?        00:00:01 redis-server *:7000
root     16094     1  0 22:53 ?        00:00:01 redis-server *:7001
root     16499     1  0 22:53 ?        00:00:01 redis-server *:7002
root     33120     1  0 23:25 ?        00:00:00 redis-sentinel *:26379 [sentinel]
root     33663     1  0 23:25 ?        00:00:00 redis-sentinel *:26380 [sentinel]
root     33981     1  0 23:25 ?        00:00:00 redis-sentinel *:26381 [sentinel]
```

Jedis客户端测试

```
package com.hefery;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import redis.clients.jedis.Jedis;
import redis.clients.jedis.JedisSentinelPool;

import java.security.PrivateKey;
import java.util.HashSet;
import java.util.Random;
import java.util.Set;
import java.util.concurrent.TimeUnit;
```

```
/**
 * @Author：Hefery
 * @Version：1.0.0
 * @Date：2021/10/12 23:43
 * @Description：
 */
public class RedisSentinelFailoverTest {

    private static Logger logger =  LoggerFactory.getLogger(RedisSentinelFailoverTest.class);

    public static void main(String[] args) {

        String masterName = "mymaster";
        Set<String> sentinels = new HashSet<String>();
        sentinels.add("192.168.1.13:26379");
        sentinels.add("192.168.1.13:26380");
        sentinels.add("192.168.1.13:26381");
        JedisSentinelPool jedisSentinelPool = new JedisSentinelPool(masterName, sentinels);

        int counter = 0;

        while (true) {
            counter++;
            Jedis jedis = null;
            try {
                jedis = jedisSentinelPool.getResource();
                int index = new Random().nextInt(100000);
                String key = "k-" + index;
                String value = "v-" + index;
                jedis.set(key, value);
                if (counter % 100 == 0) {
                    logger.info("{} value is {}", key, jedis.get(key));
                }
                TimeUnit.MICROSECONDS.sleep(10);
            } catch (Exception e) {
                logger.error(e.getMessage(), e);
            } finally {
                if (jedis != null) {
                    jedis.close();
                }
            }
        }

    }

}
```

定时任务

1. 每10秒每个sentinel对master和slave执行info

发现slave节点

确认主从关系

2. 每2秒每个dentine通过master节点的channel交换信息（pub/sub）

通过_sentinel_:hell频道交互

交互对节点的“看法”和自身信息

3. 每1秒每个sentinel对其他sentinel和reds执行ping

Redis Cluster集群

数据分布

实现：

- 顺序分区：[1-100] -> [1-33]+[34-66]+[67-100]
- 哈希分区：hash(key)%nodes

分布形式	特点	产品
顺序分区	数据分散度高；键值分布与业务无关，无法顺序访问；支持批量操作	Redis Cluster
哈希分区	数据分散易倾斜；键值分布与业务相关，可顺序访问；支持批量操作	HBase，BigTable

哈希分区：

节点取余分区：hash(key)%nodes
    3 node（hash(key)%3）：[1-100] -> [3 6 9...]+[1 4...100]+[2 5...98]
    4 node（hash(key)%4）：[1-100] -> [4 8...100]+[1 5...97]+[2 6...98]+[3 7...99]
    扩容：
        添加单个：数据迁移80%
        倍数扩容：数据迁移50%
一致性哈希分区：token顺时针分配
    节点伸缩：只影响临近节点，还是有数据迁移
    翻倍伸缩：保证最小迁移数据和负载均衡
虚拟槽分区：
    预设虚拟槽：每个槽映射一个数据子集，一般比节点数大
    良好的哈希函数：CRC16
    服务端管理节点、槽、数据

搭建集群

集群架构：
    装备节点：
    meet：沟通(连通图)，节点共享消息
    指派槽：给节点指派槽
    主从复制：

原生命令

装备节点

```
port ${port}
daemonize yes
dir /home/hefery/Downloads/Redis/redis/data/log
dbfilename "dump-cluster-${port}.rdb"
logfile "redis-cluster-${port}.log"
cluster-enabled yes
cluster-config-file redis-cluster-nodes-${port}.conf
cluster-node-timeout 15000
cluster-require-full-coverage no

sed 's/7000/7001/g' redis-cluster-7000.conf >  redis-cluster-7001.conf
sed 's/7000/7002/g' redis-cluster-7000.conf >  redis-cluster-7002.conf
sed 's/7000/7003/g' redis-cluster-7000.conf >  redis-cluster-7003.conf
sed 's/7000/7004/g' redis-cluster-7000.conf >  redis-cluster-7004.conf
sed 's/7000/7005/g' redis-cluster-7000.conf >  redis-cluster-7005.conf

redis-server redis-cluster-7000.conf
redis-server redis-cluster-7001.conf
redis-server redis-cluster-7002.conf
redis-server redis-cluster-7003.conf
redis-server redis-cluster-7004.conf
redis-server redis-cluster-7005.conf

# 查看cluster启动情况
> ps -ef | grep redis
root      2438     1  0 23:17 ?        00:00:00 redis-server *:7000 [cluster]
root      3141     1  0 23:18 ?        00:00:00 redis-server *:7001 [cluster]
root      3395     1  0 23:18 ?        00:00:00 redis-server *:7002 [cluster]
root    114738     1  0 23:11 ?        00:00:00 redis-server *:7003 [cluster]
root    115012     1  0 23:11 ?        00:00:00 redis-server *:7004 [cluster]
root    115527     1  0 23:11 ?        00:00:00 redis-server *:7005 [cluster]
# 查看cluster info信息
> redis-cli -p 7000 cluster info
cluster_state:fail
cluster_slots_assigned:0
cluster_slots_ok:0
cluster_slots_pfail:0
cluster_slots_fail:0
cluster_known_nodes:1
cluster_size:0
cluster_current_epoch:0
cluster_my_epoch:0
cluster_stats_messages_sent:0
cluster_stats_messages_received:0
```

meet

```
redis-cli -p 7000 cluster meet 127.0.0.1 7001
redis-cli -p 7000 cluster meet 127.0.0.1 7002
redis-cli -p 7000 cluster meet 127.0.0.1 7003
redis-cli -p 7000 cluster meet 127.0.0.1 7004
redis-cli -p 7000 cluster meet 127.0.0.1 7005

> redis-cli -p 7005 cluster nodes
```

```
528be9a68ce532a31b838724da23646834c2abf4 127.0.0.1:7003 master - 0 1634140138180 3
onnected
523d5b8cfd4e32a65629ff6acd09e933966448ec 127.0.0.1:7004 master - 0 1634140139196 4 c
nnected
6096428be6d89c2c5d43a2125a766fc88a77ba78 127.0.0.1:7001 master - 0 1634140135134 0
onnected
10b0195040a76686ebb222ffa363d45bcf4a3de0 127.0.0.1:7000 master - 0 1634140137164 1 c
nnected
326389de2601f42cc48fdd3151c4629b5c3dcdc3 127.0.0.1:7005 myself,master - 0 0 5 connect
d
2ac8fdc0854b74502bf0b284cf37bd8c82a06d79 127.0.0.1:7002 master - 0 1634140136148 2 c
nnected

# 查看cluster info信息
> redis-cli -p 7000 cluster info
cluster_state:fail
cluster_slots_assigned:0
cluster_slots_ok:0
cluster_slots_pfail:0
cluster_slots_fail:0
cluster_known_nodes:6
cluster_size:0
cluster_current_epoch:5
cluster_my_epoch:1
cluster_stats_messages_sent:2306
cluster_stats_messages_received:2306
```

指派槽

```
使用脚本：vim addslots.sh
start=$1
end=$2
port=$3
for slot in `seq ${start} ${end}`
do
  echo "slot:${slot}"
  redis-cli -p ${port} cluster addslots ${slot}
done

sh addslots.sh 0 5461 7000
sh addslots.sh 5462 10922 7001
sh addslots.sh 10923 16383 7002

# 查看cluster info信息
> redis-cli -p 7005 cluster nodes
528be9a68ce532a31b838724da23646834c2abf4 127.0.0.1:7003 master - 0 1634141761402 3
onnected
523d5b8cfd4e32a65629ff6acd09e933966448ec 127.0.0.1:7004 master - 0 1634141758380 4 c
nnected
6096428be6d89c2c5d43a2125a766fc88a77ba78 127.0.0.1:7001 master - 0 1634141763418 0
onnected 5462-10922
10b0195040a76686ebb222ffa363d45bcf4a3de0 127.0.0.1:7000 master - 0 1634141760394 1 c
nnected 0-5461
326389de2601f42cc48fdd3151c4629b5c3dcdc3 127.0.0.1:7005 myself,master - 0 0 5 connect
```

d
2ac8fdc0854b74502bf0b284cf37bd8c82a06d79 127.0.0.1:7002 master - 0 1634141762410 2 c
nnected 10923-16383

主从复制

从：7003 主：7000
    # redis-cli -p 7003 cluster replicate ${node-id-7000}
    redis-cli -p 7003 cluster replicate 10b0195040a76686ebb222ffa363d45bcf4a3de0
从：7003 主：7000
    # redis-cli -p 7004 cluster replicate ${node-id-7001}
    redis-cli -p 7004 cluster replicate 6096428be6d89c2c5d43a2125a766fc88a77ba78
从：7003 主：7000
    # redis-cli -p 7005 cluster replicate ${node-id-7002}
    redis-cli -p 7005 cluster replicate 2ac8fdc0854b74502bf0b284cf37bd8c82a06d79

查看cluster info信息
> redis-cli -p 7000 cluster nodes
528be9a68ce532a31b838724da23646834c2abf4 127.0.0.1:7003 slave 10b0195040a76686ebb
22ffa363d45bcf4a3de0 0 1634142192928 3 connected
6096428be6d89c2c5d43a2125a766fc88a77ba78 127.0.0.1:7001 master - 0 1634142194947 0
onnected 5462-10922
2ac8fdc0854b74502bf0b284cf37bd8c82a06d79 127.0.0.1:7002 master - 0 1634142195956 2 c
nnected 10923-16383
326389de2601f42cc48fdd3151c4629b5c3dcdc3 127.0.0.1:7005 slave 2ac8fdc0854b74502bf0b
84cf37bd8c82a06d79 0 1634142193938 5 connected
523d5b8cfd4e32a65629ff6acd09e933966448ec 127.0.0.1:7004 slave 6096428be6d89c2c5d43
2125a766fc88a77ba78 0 1634142191916 4 connected
10b0195040a76686ebb222ffa363d45bcf4a3de0 127.0.0.1:7000 myself,master - 0 0 1 connect
d 0-5461

查看cluster slot信息
> redis-cli -p 7000 cluster slots
1) 1) (integer) 5462
   2) (integer) 10922
   3) 1) "127.0.0.1"
      2) (integer) 7001
   4) 1) "127.0.0.1"
      2) (integer) 7004
2) 1) (integer) 10923
   2) (integer) 16383
   3) 1) "127.0.0.1"
      2) (integer) 7002
   4) 1) "127.0.0.1"
      2) (integer) 7005
3) 1) (integer) 0
   2) (integer) 5461
   3) 1) "127.0.0.1"
      2) (integer) 7000
   4) 1) "127.0.0.1"
      2) (integer) 7003

Ruby

下载编译安装Ruby

```
wget https://cache.ruby-lang.org/pub/ruby/2.3/ruby-2.3.1.tar.gz
tar -xvf ruby-2.3.1.tar.gz
cd ruby-2.3.1
./configure #-prefix=/usr/local/ruby
make && make install

cd bin/usr/local/rub
cp bin/ruby/usr/local/bin
cp bin/gem /usr/local/bin

# 查看ruby版本
ruby -v
```

安装rubygem redis

```
wget http://rubygems.org/downloads/redis-3.3.0.gem
gem install -l redis-3.3.0.gem
gem list -- check redis gem
```

安装redis-trib.rb

```
# cp $(REDIS_ HOME)/src/redis-trib. rb /usr/local/bin

> cd Downloads/Redis/redis/src
> ./redis-trib.rb
Usage: redis-trib <command> <options> <arguments ...>

  create          host1:port1 ... hostN:portN
                  --replicas <arg>
  check           host:port
  info            host:port
  fix             host:port
                  --timeout <arg>
  reshard         host:port
                  --from <arg>
                  --to <arg>
                  --slots <arg>
                  --yes
                  --timeout <arg>
                  --pipeline <arg>
  rebalance       host:port
                  --weight <arg>
                  --auto-weights
                  --use-empty-masters
                  --timeout <arg>
                  --simulate
                  --pipeline <arg>
                  --threshold <arg>
  add-node        new_host:new_port existing_host:existing_port
                  --slave
                  --master-id <arg>
```

```
  del-node       host:port node_id
  set-timeout    host:port milliseconds
  call           host:port command arg arg .. arg
  import         host:port
                 --from <arg>
                 --copy
                 --replace
  help           (show this help)

For check, fix, reshard, del-node, set-timeout you can specify the host and port of any workin
 node in the cluster.
```

准备节点

```
port 7000
daemonize yes
dir /home/hefery/Downloads/Redis/redis/data/log
dbfilename "dump-cluster-ruby-7000.rdb"
logfile "redis-cluster-ruby-7000.log"
cluster-enabled yes
cluster-config-file redis-cluster-ruby-nodes-7000.conf
cluster-require-full-coverage no


sed 's/7000/7001/g' redis-cluster-ruby-7000.conf > redis-cluster-ruby-7001.conf
sed 's/7000/7002/g' redis-cluster-ruby-7000.conf > redis-cluster-ruby-7002.conf
sed 's/7000/7003/g' redis-cluster-ruby-7000.conf > redis-cluster-ruby-7003.conf
sed 's/7000/7004/g' redis-cluster-ruby-7000.conf > redis-cluster-ruby-7004.conf
sed 's/7000/7005/g' redis-cluster-ruby-7000.conf > redis-cluster-ruby-7005.conf

redis-server redis-cluster-ruby-7000.conf
redis-server redis-cluster-ruby-7001.conf
redis-server redis-cluster-ruby-7002.conf
redis-server redis-cluster-ruby-7003.conf
redis-server redis-cluster-ruby-7004.conf
redis-server redis-cluster-ruby-7005.conf

> ps -ef | grep redis-server
root      72881      1  0 01:25 ?        00:00:00 redis-server *:7000 [cluster]
root      73203      1  0 01:25 ?        00:00:00 redis-server *:7001 [cluster]
root      73436      1  0 01:25 ?        00:00:00 redis-server *:7002 [cluster]
root      73756      1  0 01:25 ?        00:00:00 redis-server *:7003 [cluster]
root      73992      1  0 01:25 ?        00:00:00 redis-server *:7004 [cluster]
root      74733      1  0 01:26 ?        00:00:00 redis-server *:7005 [cluster]

# redia路径下的redis-trib.rb
cd Downloads/Redis/redis/src
# 使用create创建nodes(选yes)
./redis-trib.rb create --replicas 1 127.0.0.1:7000 127.0.0.1:7001 127.0.0.1:7002 127.0.0.1:7003 1
7.0.0.1:7004 127.0.0.1:7005
```

集群扩容

加入集群作用：

- 为它迁移槽和数据实现扩容
- 作为从节点负责故障转移

准备新节点

```
port ${port}
daemonize yes
dir /home/hefery/Downloads/Redis/redis/data/log
dbfilename "dump-cluster-${port}.rdb"
logfile "redis-cluster-${port}.log"
cluster-enabled yes
cluster-config-file redis-cluster-nodes-${port}.conf
cluster-node-timeout 15000
cluster-require-full-coverage no

sed 's/7000/7006/g' redis-cluster-7000.conf >  redis-cluster-7006.conf
sed 's/7000/7007/g' redis-cluster-7000.conf >  redis-cluster-7007.conf

redis-server redis-cluster-7006.conf
redis-server redis-cluster-7007.conf
```

加入集群

```
redis-cli -p 7000 cluster meet 127.0.0.1 7006
redis-cli -p 7000 cluster meet 127.0.0.1 7007
```

主从复制

```
# 7007 replicate 7006
redis-cli -p 7007 cluster replicate 6a0d5b1f102b16ff88a0b621bc2f037798540689

redis-cli -p 7000 cluster nodes
bc125b4301176b75a20b77da37ce5a5111619ccf 127.0.0.1:7004 slave d5f6c8b25537422fde987
4c802d9fb59a07ff94 0 1634223198639 5 connected
c19b6520969c1fc4c0e50475fe73fa1a8ec54fda 127.0.0.1:7002 master - 0 1634223200660 3 co
nected 10923-16383
4aabd108347571d1123778bfeaddcd7a8e602194 127.0.0.1:7007 slave 6a0d5b1f102b16ff88a0
621bc2f037798540689 0 1634223199650 7 connected
6830085d334270a61508afb3c96e4f6648efb02f 127.0.0.1:7003 slave 49f14b591968ed22462b3
34262f7dcb9e6a8b92 0 1634223201670 4 connected
2163c9c61d776d5f266c1ab687f7cefc7f209feb 127.0.0.1:7005 slave c19b6520969c1fc4c0e504
5fe73fa1a8ec54fda 0 1634223195612 6 connected
49f14b591968ed22462b3a34262f7dcb9e6a8b92 127.0.0.1:7000 myself,master - 0 0 1 connect
d 0-5460
d5f6c8b25537422fde987f4c802d9fb59a07ff94 127.0.0.1:7001 master - 0 1634223197635 2 co
nected 5461-10922
6a0d5b1f102b16ff88a0b621bc2f037798540689 127.0.0.1:7006 master - 0 1634223199144 0 c
nnected
```

迁移槽和数据

1.对目标节点发送：cluster setslot{slot} importing {source Nodeld}命令，让目标节点准备导入槽

数据
2.对于源节点发送：cluster setslot {slot} migrating {targetNodeId}命令，让源节点准备迁出槽的据
3.源节点循环执行：cluster getkeysinslot {slot}{count}命令，每次获取 count个属于槽的键
4.在源节点上执行：migrate {targetIP} {targetPort} key 0 {timeout}命令把指定key迁移
5.重复执行步骤3~4直到槽下所有的键数据迁移到目标节点
6.向集群内所有主节点发送 cluster setslot{slot} node {targetNodeId}命令，通知槽分配给目标节点

```
> ./redis-trib.rb reshard 127.0.0.1:7000
How many slots do you want to move (from 1 to 16384)? 4092
What is the receiving node ID? 6a0d5b1f102b16ff88a0b621bc2f037798540689
Please enter all the source node IDs.
  Type 'all' to use all the nodes as source nodes for the hash slots.
  Type 'done' once you entered all the source nodes IDs.
Source node #1:all
Do you want to proceed with the proposed reshard plan (yes/no)? yes

# 查看槽的分配
> redis-cli -p 7000 cluster nodes
bc125b4301176b75a20b77da37ce5a5111619ccf 127.0.0.1:7004 slave d5f6c8b25537422fde987
4c802d9fb59a07ff94 0 1634223721205 5 connected
c19b6520969c1fc4c0e50475fe73fa1a8ec54fda 127.0.0.1:7002 master - 0 1634223719191 3 co
nected 12286-16383
4aabd108347571d1123778bfeaddcd7a8e602194 127.0.0.1:7007 slave 6a0d5b1f102b16ff88a0
621bc2f037798540689 0 1634223721205 8 connected
6830085d334270a61508afb3c96e4f6648efb02f 127.0.0.1:7003 slave 49f14b591968ed22462b3
34262f7dcb9e6a8b92 0 1634223714154 4 connected
2163c9c61d776d5f266c1ab687f7cefc7f209feb 127.0.0.1:7005 slave c19b6520969c1fc4c0e504
5fe73fa1a8ec54fda 0 1634223716677 6 connected
49f14b591968ed22462b3a34262f7dcb9e6a8b92 127.0.0.1:7000 myself,master - 0 0 1 connect
d 1363-5460
d5f6c8b25537422fde987f4c802d9fb59a07ff94 127.0.0.1:7001 master - 0 1634223717180 2 co
nected 6826-10922
6a0d5b1f102b16ff88a0b621bc2f037798540689 127.0.0.1:7006 master - 0 1634223720197 8 c
nnected 0-1362 5461-6825 10923-1228
```

收缩集群

```
# 迁移槽：7006 -> 7000
> ./redis-trib.rb reshard --from 6a0d5b1f102b16ff88a0b621bc2f037798540689 --to 49f14b59
968ed22462b3a34262f7dcb9e6a8b92 --slots 1366 127.0.0.1:7006
Do you want to proceed with the proposed reshard plan (yes/no)? yes

# 迁移槽：7006 -> 7001
> ./redis-trib.rb reshard --from 6a0d5b1f102b16ff88a0b621bc2f037798540689 --to d5f6c8b2
537422fde987f4c802d9fb59a07ff94 --slots 1366 127.0.0.1:7006
Do you want to proceed with the proposed reshard plan (yes/no)? yes

# 迁移槽：7006 -> 7002
> ./redis-trib.rb reshard --from 6a0d5b1f102b16ff88a0b621bc2f037798540689 --to c19b6520
69c1fc4c0e50475fe73fa1a8ec54fda --slots 1366 127.0.0.1:7006

# 查看7006槽以及迁移完毕
> redis-cli -p 7000 cluster nodes
```

```
bc125b4301176b75a20b77da37ce5a5111619ccf 127.0.0.1:7004 slave d5f6c8b25537422fde987
4c802d9fb59a07ff94 0 1634224663138 10 connected
c19b6520969c1fc4c0e50475fe73fa1a8ec54fda 127.0.0.1:7002 master - 0 1634224664646 11 c
nnected 10927-16383
4aabd108347571d1123778bfeaddcd7a8e602194 127.0.0.1:7007 slave c19b6520969c1fc4c0e5
475fe73fa1a8ec54fda 0 1634224665149 11 connected
6830085d334270a61508afb3c96e4f6648efb02f 127.0.0.1:7003 slave 49f14b591968ed22462b3
34262f7dcb9e6a8b92 0 1634224663640 9 connected
2163c9c61d776d5f266c1ab687f7cefc7f209feb 127.0.0.1:7005 slave c19b6520969c1fc4c0e504
5fe73fa1a8ec54fda 0 1634224662132 11 connected
49f14b591968ed22462b3a34262f7dcb9e6a8b92 127.0.0.1:7000 myself,master - 0 0 9 connect
d 0-5463
d5f6c8b25537422fde987f4c802d9fb59a07ff94 127.0.0.1:7001 master - 0 1634224666154 10 c
nnected 5464-10926
6a0d5b1f102b16ff88a0b621bc2f037798540689 127.0.0.1:7006 master - 0 1634224664143 8 c
nnected

# 下线：7006+7007
> ./redis-trib.rb del-node 127.0.0.1:7000 6a0d5b1f102b16ff88a0b621bc2f037798540689
>>> Removing node 6a0d5b1f102b16ff88a0b621bc2f037798540689 from cluster 127.0.0.1:7
00
>>> Sending CLUSTER FORGET messages to the cluster...
>>> SHUTDOWN the node.
> ./redis-trib.rb del-node 127.0.0.1:7000 4aabd108347571d1123778bfeaddcd7a8e602194
>>> Removing node 4aabd108347571d1123778bfeaddcd7a8e602194 from cluster 127.0.0.1:
000
>>> Sending CLUSTER FO
```

客户端路由

moved重定向：槽已经确定迁移

ask重定向：槽还在迁

smart客户端：JedisCluster

原理：

- 从集群中选一个可运行节点，使用cluster slots初始化槽和节点映射
- 将cluster slots的结果映射到本地，为每个节点创建JedisPool
- 准备执行命令

开发运维问题

集群完整性

cluster-require-full-coverage：
集群中16384个槽全部可用：保证集群完整性
节点故障或者正在故障转移：（error） CLUSTERDOWN The cluster is down
建议：cluster-require-full-coverage no

带宽消耗

避免“大”集群：避免多业务使用一个集群，大业务可以多集群
cluster-node-timeout：带宽和故障转移速度的均衡
    cluster-node-timeout=15000，ping/pong带宽为25Mb
    cluster-node-timeout=20000，ping/pong带宽低于15Mb
尽量均匀分配到多机器上：保证高可用和带宽

Pub/Sub广播

问题：publish在集群每个节点广播：加重带宽
解决：单独“走”一套 Redis Sentinel

数据倾斜

数据倾斜：内存不均
    节点和槽分配不均：
        redis-trib.rb info ip:port 查看节点、槽、键值分布
        redis-trib.rb rebalance ip:port 进行均衡（谨慎使用）
    不同槽对应键值数量差异较大:
        CRC16正常情况下比较均匀
        可能存在hash_tag
        cluster countkeysinslot {slot} 获取槽对应键值个数
    包含bigkey:
        bigley：例如大字符串、几百万的元素的hash、set等
        从节点：redis-cl- bigkeys
        优化：优化数据结构
    内存相关配置不一致:
        hash-max-ziplist-value, set-max-intset-entriesg
        优化：定期“检查”配置一致性

请求倾斜：热点
    热点key：重要的key或bigkey
    优化：
        避免 bigley
        热键不要用hash_tag
        当一致性不高时，可以用本地缓存+MQ

读写分离

只读连接：集群模式的从节点不接受任何读写请求
    - 重定向到负责槽的主节点
    - readonly命令可以读：连接级别命令
读写分离：更加复杂
    同样的问题：复制延迟、读取过期数据、从节点故障
    修改客户端：cluster slaves {nodeId}

数据迁移

离线/在线迁移
    官方迁移工具：redis- trib. rb import
    只能从单机迁移到集群
    不支持在线迁移：source需要停写
    不支持断点续传

单线程迁移：影响速度
在线迁移：
唯品会 redis-migrate-tool
豌豆荚：redis-port

缓存的使用与设计

缓存好处：

- 加速读写(通过缓存加速读写速度)：CPUL1/L2/L3 Cache、 Linux page Cache加速硬盘读写、浏器缓存、 Ehcache缓存数据库结果
- 降低后端负载(后端服务器通过前端缓存降低负载)：业务端使用 Redis降低后端MySQ负载等

缓存成本：

- 数据不一致：缓存层和数据层有时间窗口不一致，和更新策略有关
- 代码维护成本：多了一层缓存逻辑
- 运维成本：例如Redis Cluster

使用场景：

- 降低后端负载：对高消耗的SQL:join结果集/分组统计结果缓存
- 加速请求：响应利用Redis/ Memcache优化IO响应时间
- 大量写合并为批量写：如计数器先 Redis累加再批量写DB

缓存更新策略

LRU/LFU/FIFO算法剔除：例如 maxmemory-policy

超时剔除：例如expire

主动更新：开发控制生命周期

建议：

低一致性：最大内存和淘汰策略

高一致性 : 超时剔除和主动更新结合，最大内存和淘汰策略兜底

策略	一致性	维护成本
LRU/LFU/FIFO算法剔除	最差	低
超时剔除	较差	低
主动更新	强	高

缓存粒度控制

缓存粒度控制角度：

通用性：全量属性更好

占用空间：部分属性更好

代码维护：表面上全量属性更好

缓存穿透优化

缓存穿透问题：大量请求不命中 request -> cache(miss) -> storage(miss) -> request(return null)

原因：

业务代码自身问题

恶意攻击、爬虫等等

发现：

业务的相应时间

业务本身问题

相关指标：总调用数、缓存层命中数、存储层命中数

解决缓存空对象：request -> cache(miss) -> storage(miss) -> cache(return null) + request(retu
n null)

问题：

需要更多的键

缓存层和存储层数据“短期”不一致

解决：

布隆过滤器拦截

缓存雪崩优化

由于cache服务承载大量请求，当cache服务异常/脱机，流量直接压向后端组件例如DB，造成级联故障

优化：Redis Cluster、Redis Sentinel、主从漂移

保证缓存高可用性：个别节点、个别机器、甚至是机房，例如Redis Cluster、Redis Sentinel、VIP

依赖隔离组件为后端限流：依赖隔离组件——线程池/信号量隔离组件HYSTRIX

提前演练：例如压力测试

无底洞问题优化

问题描述：

2010年， Facebook有了3000个 Memcache节点

发现问题： “加”机器性能没能提升，反而下降

问题关键：

更多的机器不代表更高的性能

批量接口需求(mget, mset等)

数据增长与水平扩展需求

优化：

IO优化：

命令本身优化：例如慢查询keys、 hgetall bigkey

减少网络通信次数

降低接入成本：例如客户端长连接/连接池、NIO等

批量优化的方法：

串行mget

串行IO

并行IO

hash_tag

方案	优点	缺点	网络IO
串行mget	编程简单，少量keys满足需求	大量key请求延迟严重	O(keys)
串行IO	编程简单，少量节点满足需求	大量node延迟严重	O(node)
并行IO	利用并行特性，延迟取决于最慢的节点	编复杂，超时定位问题难	O(max_slow(node))
hash_tag	性能最高	读写增加tag维护成本，tag布易出现数据倾斜	O(1)

热点key重建优化

可题描述：热点key+较长的重建时间

目标：

减少重缓存的次数

数据尽可能一致

减少潜在危险

解决：

互斥锁：

永远不过期：

缓存层面：没有设置过期时间没有用 expire

功能层面：为每个value添加逻辑过期时间，但发现超过逻辑过期时间后，会使用单独的线程去构建存

方案	优点	缺点
互斥锁	思路简单，保证一致性	码复杂度增加，存在死锁的风险
永远不过期	基本杜绝热点key重建问题	保证一致性，逻辑过期时间增加维护成本和内存成本

基于Redis的分布式布隆过滤器

布隆过滤器原理

实现原理：一个很长的二进制向量和若干个哈希函数

布隆过滤器构建：

参数：m个二进制向量，n个预备数据，k个hash函数

布隆过滤器误差率

肯定存在误差：恰好都命中了

直观因素：m/n的比率，hash函数的个数

m/n与误差率成反比，k与误差率成反比

Redis开发规范

Key名设计

可读性和管理性：以业务名（或数据库名）为前缀（防止key冲突），用冒号分割可
  如：业务名表名id，如：ugc:video:1

简洁性：保证语义的前提下，控制key的长度，当key较多时，内存占用也不容忽视（reds3:39字节ebstr)
  如：user:{uid}:friends:messages:{mid} -> u:{uid}:fr:m:{mid}

不要包含特殊字符：反例：包含空格、换行、单双引号以及其他转义字符

Value设计

bigley:
  强制:
    1.string类型控制在10KB以内
    2.hash、list、set、zset元素个数不要超过5000
  反例：一个包含几百万个元素的list、hash等，一个巨大的json字符串
  危害:
    1.网络阻塞
    2.Redis阻塞：慢查询，hgetall、lrange、zgrange
    3.集群节点数据不均衡
    4.频繁序列化：应用服务器CPU消耗
      反序列消耗:
        Redis客户端本身不负责序列化
        应用频繁序列化和反序列化 bigley：本地缓存或 Redis缓存
  发现:
    1.应用异常
    2.redis-ci --bigkeys
    3.scan + debug object
    4.主动报警：网络流量监控、客户端监控
    5.内核热点key问题优化
  删除:
    阻塞：注意隐性删除（过期、 rename等）
    Reds4.0：lazy delete（ unlink命令）

优化：
优化数据结构：例如二级拆分
物理隔离或者万兆网卡：不是治标方案
命令优化：例如hgetall --> hmget、scan
报警和定期优化

Redis面试

对于Redis的理解

Redis是完全开源免费的一个高性能的 key-value 数据库

特点：

- Redis 支持数据的持久化，可以将内存中的数据保存在磁盘中，重启的时候可以再次加载进行使用
- Redis 不仅仅支持简单的 key-value 类型的数据，同时还提供 list，set，zset，hash 等数据结构存储
- Redis 支持数据的备份，即 master-slave 模式的数据备份

Redis 优势

- 性能极高：Redis 能读的速度是 110000 次/s,写的速度是 81000 次/s（查找和操作时间复杂度都是O1）
- 丰富的数据类型：Redis 支持二进制案例的 string，list，set，zset，hash 数据类型操作
- 原子：Redis 的所有操作都是原子性的，意思就是要么成功执行要么失败完全不执行。单个操作是子性的。多个操作也支持事务，即原子性，通过 MULTI 和 EXEC指令包起来。
- 丰富的特性：Redis 还支持 publish/subscribe, 通知, key 过期等等特性

为什么要使用Redis

安全，解决高并发

为什么Redis需要把所有数据放到内存中?

Redis 为了达到最快的读写速度将数据都读到内存中，并通过异步的方式将数据写入磁盘。所以 redis 具有快速和数据持久化的特征。如果不将数据放在内存中，磁盘 I/O 速度为严重影响 redis 的性能。内存越来越便宜，redis会越来越受欢迎。如果设置了最大使用的内存，则数据已有记录数达到内存限后不能继续插入新值

一个 Redis 实例最多能存放多少的 keys?

理论上 Redis 可以处理多达 232 的 keys，并且在实际中进行了测试，每个实例至少存放了 2 亿 5 千的 keys。我们正在测试一些较大的值。任何 list、set、和 sorted set 都可以放 232 个元素。换句话，Redis 的存储极限是系统中的可用内存值

为什么要用Redis，相比于如Memcached,MongoDB有什么优势

Memcached 所有的值均是简单的字符串，redis 作为其替代者，支持更为丰富的数据类型

Redis 的速度比 Memcached 快

Redis 可以持久化其数据

存储方式 Memecache 把数据全部存在内存之中，断电后会挂掉，数据不能超过内存大小。 Redis 部份存在硬盘上，这样能保证数据的持久性

Redis怎么做到多路IO复用

Redis的原子性实现

对于Redis而言，命令的原子性指的是：一个操作的不可以再分，操作要么执行，要么不执行

Redis的操作之所以是原子性的，是因为Redis是单线程的

Redis本身提供的所有API都是原子操作，Redis中的事务其实是要保证批量操作的原子性

Redis在项目中进行怎么样的使用，项目中redis主要存储什么

热点数据和冷数据

热点数据，缓存才有价值

对于冷数据而言，大部分数据可能还没有再次访问到就已经被挤出内存，不仅占用内存，而且价值不。频繁修改的数据，看情况考虑使用缓存

寿星列表、导航信息都存在一个特点，就是信息修改频率不高，读取通常非常高的场景

对于热点数据，比如我们的某IM产品，生日祝福模块，当天的寿星列表，缓存以后可能读取数十万次再举个例子，某导航产品，我们将导航信息，缓存以后可能读取数百万次

数据更新前至少读取两次，缓存才有意义，如果缓存还没有起作用就失效了，那就没有太大价值了

存在修改频率很高，但是又不得不考虑缓存的场景，比如，这个读取接口对数据库的压力很大，但是是热点数据，这个时候就需要考虑通过缓存手段，减少数据库的压力，比如我们的某助手产品的，点数，收藏数，分享数等是非常典型的热点数据，但是又不断变化，此时就需要将数据同步保存到Redi缓存，减少数据库压力

Redis的单线程为什么也能这么快?

完全基于内存

单线程避免不必要的上下文切换、CPU消耗

采用了非阻塞I/O多路复用机制

Redis底层数据结构？详细说string底层的sds

讲一下zset跳表?

Redis常用数据结构，常用的指令

string（字符串），hash（哈希），list（列表），set（集合）及 zsetsorted set（有序集合）

实际项目中比较常用的是 string，hash

Redis 中高级用户，还需要加上下面几种数据结构 HyperLogLog、Geo、Pub/Sub

Redis Module，像 BloomFilter，RedisSearch，Redis-ML，面试官得眼睛就开始发亮了

一个字符串类型的值能存储最大容量是多少?

512M

Redis数据过期

Redis的过期策略以及内存淘汰机制

定期删除+惰性删除

为什么不用定时删除策略?

定时删除,用一个定时器来负责监视key,过期则自动删除。虽然内存及时释放，但是十分消耗CPU资源在大并发请求下，CPU要将时间应用在处理请求，而不是删除key,因此没有采用这一策略

定期删除+惰性删除是如何工作的呢?

定期删除，redis默认每个100ms检查，是否有过期的key,有过期key则删除。需要说明的是，redis不每个100ms将所有的key检查一次，而是随机抽取进行检查(如果每隔100ms,全部key进行检查，redi岂不是卡死)。因此，如果只采用定期删除策略，会导致很多key到时间没有删除。于是，惰性删除派用场。也就是说在你获取某个key的时候，redis会检查一下，这个key如果设置了过期时间那么是否期了？如果过期了此时就会删除

采用定期删除+惰性删除就没其他问题了么?

不是的，如果定期删除没删除key。然后你也没即时去请求key，也就是说惰性删除也没生效。这样，rdis的内存会越来越高。那么就应该

采用内存淘汰机制

Redis 的回收策略（淘汰策略）

同：MySQL 里有 2000w 数据，redis 中只存 20w 的数据，如何保证 redis 中的数据都是热点数据?

volatile-lru：从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的数据淘汰

volatile-ttl：从已设置过期时间的数据集（server.db[i].expires）中挑选将要过期的数据淘汰

volatile-random：从已设置过期时间的数据集（server.db[i].expires）中任意选择数据淘汰

allkeys-lru：从数据集（server.db[i].dict）中挑选最近最少使用的数据淘汰

allkeys-random：从数据集（server.db[i].dict）中任意选择数据淘汰

no-enviction（驱逐）：禁止驱逐数据

Redis过期键删除策略

● 定时删除:在设置键的过期时间的同时，创建一个定时器 timer). 让定时器在键的过期时间来临时，即执行对键的删除操作

- 惰性删除:放任键过期不管，但是每次从键空间中获取键时，都检查取得的键是否过期，如果过期的，就删除该键;如果没有过期，就返回该键
- 定期删除:每隔一段时间程序就对数据库进行一次检查，删除里面的过期键。至于要删除多少过期键以及要检查多少个数据库，则由算法决定

Redis key 的过期时间和永久有效分别怎么设置?

EXPIRE 和 PERSIST 命令

Redis持久化机制

Redis支持持久化，通过持久化机制把内存中的数据同步到硬盘文件来保证数据持久化。当Redis重启通过把硬盘文件重新加载到内存，就能达到恢复数据的目的

实现：单独创建fork()一个子进程，将当前父进程的数据库数据复制到子进程的内存中，然后由子进写入到临时文件中，持久化的过程结束了，再用这个临时文件替换上次的快照文件，然后子进程退出内存释放。

RDB：Redis默认的持久化方式。按照一定的时间周期策略把内存的数据以快照的形式保存到硬盘的进制文件。即Snapshot快照存储，对应产生的数据文件为dump.rdb，通过配置文件中的save参数来义快照的周期

AOF：Redis会将每一个收到的写命令都通过Write函数追加到文件最后，类似于MySQL的binlog。当edis重启是会通过重新执行文件中保存的写命令来在内存中重建整个数据库的内容。当两种方式同时启时，数据恢复Redis会优先选择AOF恢复

缓存雪崩、缓存穿透、缓存预热、缓存更新、缓存降级

缓存雪崩

问题：由于原有缓存失效，新缓存未到期间(例如：我们设置缓存时采用了相同的过期时间，在同一时出现大面积的缓存过期)，所有原本应该访问缓存的请求都去查询数据库了，而对数据库CPU和内存造巨大压力，严重的会造成数据库宕机。从而形成一系列连锁反应，造成整个系统崩溃

解决：大多数系统设计者考虑用加锁（ 最多的解决方案）或者队列的方式保证来保证不会有大量的线对数据库一次性进行读写，从而避免失效时大量的并发请求落到底层存储系统上。还有一个简单方案时讲缓存失效时间分散开

缓存穿透

问题：用户查询数据，在数据库没有，自然在缓存中也不会有。这样就导致用户查询的时候，在缓存找不到，每次都要去数据库再查询一遍，然后返回空（相当于进行了两次无用的查询）。这样请求就过缓存直接查数据库，这也是经常提的缓存命中率问题

解决：采用布隆过滤器，将所有可能存在的数据哈希到一个足够大的bitmap中，一个一定不存在的据会被这个bitmap拦截掉，从而避免了对底层存储系统的查询压力。另外也有一个更为简单粗暴的法，如果一个查询返回的数据为空（不管是数据不存在，还是系统故障），我们仍然把这个空结果进缓存，但它的过期时间会很短，最长不超过五分钟。通过这个直接设置的默认值存放到缓存，这样第次到缓冲中获取就有值了，而不会继续访问数据库，这种办法最简单粗暴

缓存更新

问题：系统上线后，将相关的缓存数据直接加载到缓存系统。这样就可以避免在用户请求的时候，先询数据库，然后再将数据缓存的问题！用户直接查询事先被预热的缓存数据

解决：数据量不大，可以在项目启动的时候自动进行加载；定时刷新缓存

缓存预热

除了缓存服务器自带的缓存失效策略之外（Redis默认的有6中策略可供选择），我们还可以根据具体业务需求进行自定义的缓存淘汰

解决：

- 定时去清理过期的缓存，维护大量缓存的key是比较麻烦的
- 当有用户请求过来时，再判断这个请求所用到的缓存是否过期，过期的话就去底层系统得到新数据更新缓存，每次用户请求过来都要判断缓存失效，逻辑相对比较复杂

缓存降级

当访问量剧增、服务出现问题（如响应时间慢或不响应）或非核心服务影响到核心流程的性能时，仍需要保证服务还是可用，即使是有损服务。系统可以根据一些关键数据进行自动降级，也可以配置开实现人工降级

降级的最终目的是保证核心服务可用，即使是有损的。而且有些服务是无法降级的（如加入购物车、算）

Redis事务

Redis事务功能通过MULTI、EXEC、DISCARD和WATCH 四个原语实现

Redis会将一个事务中的所有命令序列化，然后按顺序执行

- redis 不支持回滚“Redis 在事务失败时不进行回滚，而是继续执行余下的命令”， 所以 Redis 的部可以保持简单且快速
- 如果在一个事务中的命令出现错误，那么所有的命令都不会执行
- 如果在一个事务中出现运行错误，那么正确的命令会被执行

- MULTI命令用于开启一个事务，它总是返回OK。 MULTI执行之后，客户端可以继续向服务器发任意多条命令，这些命令不会立即被执

行，而是被放到一个队列中，当EXEC命令被调用时，所有队列中的命令才会被执行。

- EXEC：执行所有事务块内的命令。返回事务块内所有命令的返回值，按命令执行的先后顺序排。当操作被打断时，返回空值
- 通过调用DISCARD，客户端可以清空事务队列，并放弃执行事务， 并且客户端会从事务状态中出
- WATCH 命令可以为 Redis 事务提供 check-and-set （CAS）行为。 可以监控一个或多个键，旦其中有一个键被修改（或删除），之后的事务就不会执行，监控一直持续到EXEC命令

Redis集群有了解么

Redis集群，为什么是16384个槽，槽分配的好处

https://zhuanlan.zhihu.com/p/435994841

Redis 集群会有写操作丢失吗?

Redis 并不能保证数据的强一致性

Redis 集群之间是如何复制的?

异步复制

Redis 集群最大节点个数是多少?

16384 个

怎么测试 Redis 的连通性?

使用 ping 命令

Redis 哈希槽

Redis 集群没有使用一致性 hash,而是引入了哈希槽的概念，Redis 集群有16384 个哈希槽，每个 key 通过 CRC16 校验后对 16384 取模来决定放置哪个槽，集群的每个节点负责一部分 hash 槽

Redis的编码方式

Redis的zset底层，skiplist和红黑树，缓存击穿

Redis发布订阅和直接用List的区别

Redis发布订阅和消息中间件的区别

Redis两种高可用方案介绍

Redis缓存到了过期时间是怎么删除缓存的?

了解Redis的哨兵模式吗?

Redis怎么保证和数据库一致性

数据进行改是直接改Redis还是删除Redis？如果直接修改为什么不行？(线程安全)

为什么要有第一次的删除Redis数据

Redis分布式锁

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，那会怎么样?
set 指令有非常复杂的参数，这个应该是可以同时把 setnx 和expire 合成一条指令来用

Redis基本数据结构，每个数据结构的组成
Redis的使用场景，Redis集群方案，Redis持久化原理

Redis可以当做数据库吗？为什么不能？Redis持久化优势劣势?关闭rdb如何主从复制?

Redis常见性能问题

Master 最好不要做任何持久化工作，如 RDB 内存快照和 AOF 日志文件
如果数据比较重要，某个 Slave 开启 AOF 备份数据，策略设置为每秒同步一次
为了主从复制的速度和连接的稳定性， Master 和 Slave 最好在同一个局域网内
尽量避免在压力很大的主库上增加从库
主从复制不要用图状结构，用单向链表结构更为稳定，即： Master <- Slave1 <- Slave2 <-Slave3