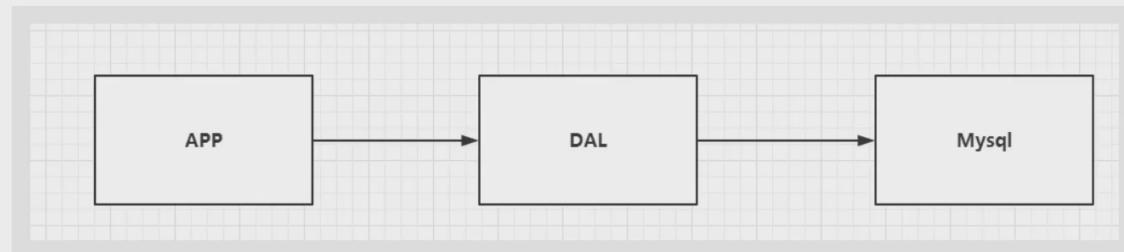


学习视频：<https://www.bilibili.com/video/BV1S54y1R7SB?p=2>

NoSQL概述

1、单机MySQL的年代！



90年代，一个基本的网站访问量一般不会太大，单个数据库完全足够！

那个时候，更多的去使用静态网页 Html ~ 服务器根本没有太大的压力！

思考一下，这种情况下：整个网站的瓶颈是什么？

1、数据量如果太大、一个机器放不下了！

2、数据的索引（B+ Tree），一个机器内存也放不下

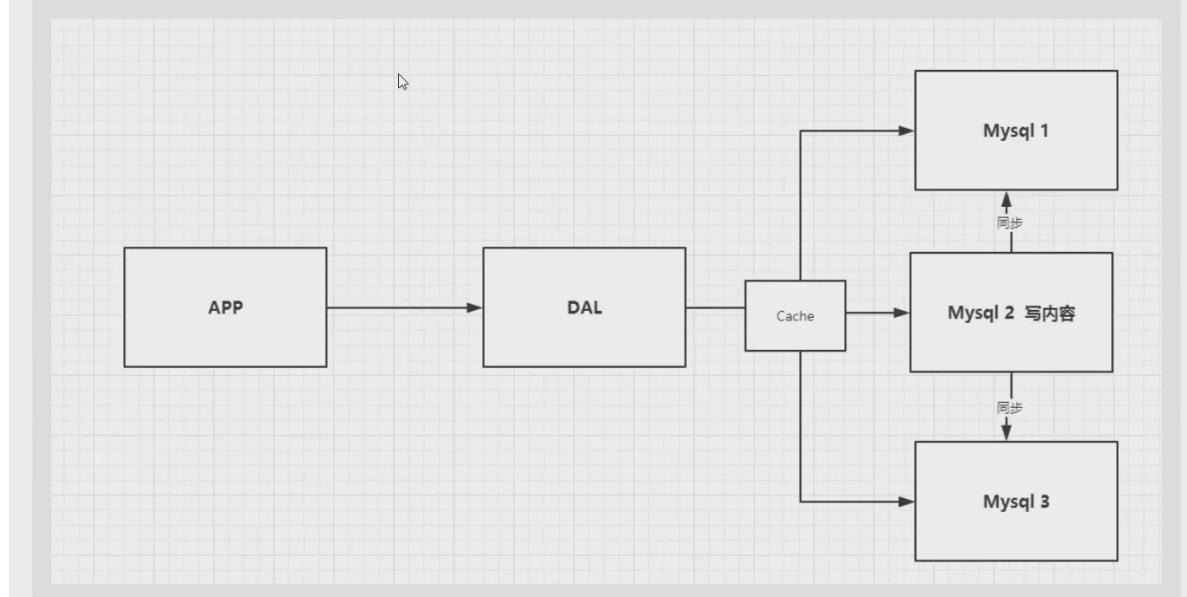
3、访问量（读写混合），一个服务器承受不了~

I

只要你开始出现以上的三种情况之一，那么你就必须晋级！

2、Memcached（缓存）+ MySQL + 垂直拆分

网站80%的情况都是在读，每次都要去查询数据库的话就十分的麻烦！所以说我们希望减轻数据的压力，我们可以使用缓存来保证效率！



3、分库分表 + 水平拆分 + MySQL集群

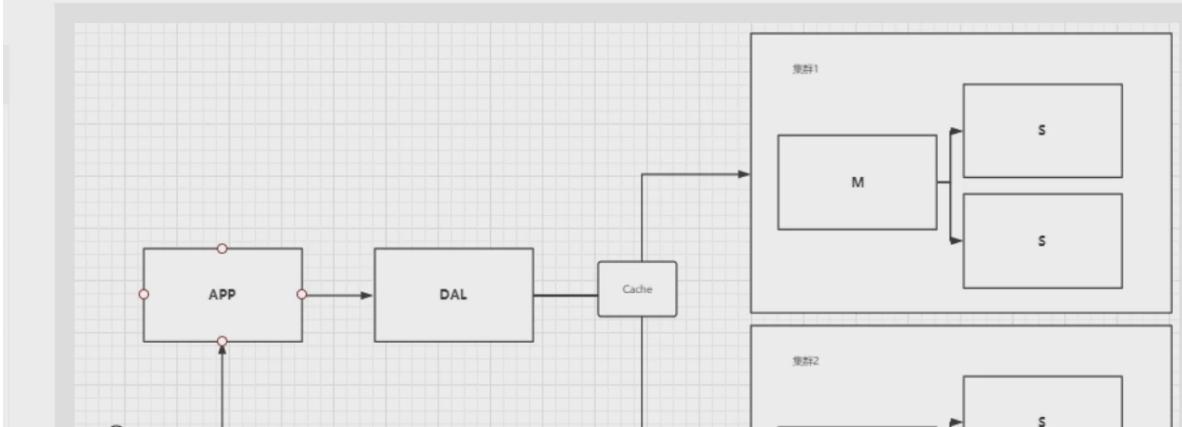
技术和业务在发展的同时，对人的要求也越来越高！

本质：数据库（读，写）

早些年MyISAM：表锁，十分影响效率！高并发下就会出现严重的锁问题

转战Innodb：行锁

慢慢的就开始使用分库分表来解决写的压力！MySQL 在哪个年代推出了表分区！



什么是 NoSQL

什么是NoSQL

NoSQL

NoSQL = Not Only SQL (不仅仅是SQL)

关系型数据库：表格，行，列

泛指非关系型数据库的，随着web2.0互联网的诞生！传统的关系型数据库很难对付web2.0时代！尤其是超大规模的高并发的社区！暴露出来很多难以克服的问题，NoSQL在当今大数据环境下发展的十分迅速，Redis是发展最快的，而且是我们当下必须要掌握的一个技术！

很多的数据类型用户的个人信息，社交网络，地理位置。这些数据类型的存储不需要一个固定的格式！不需要多月的操作就可以横向扩展的！Map<String, Object> 使用键值对来控制！

NoSQL的四大分类

NoSQL的四大分类

KV键值对：

- 新浪：Redis
- 美团：Redis + Tair
- 阿里、百度：Redis + memecache

文档型数据库（bson格式 和json一样）：

- MongoDB（一般必须要掌握）
 - MongoDB 是一个基于分布式文件存储的数据库，C++ 编写，主要用来处理大量的文档！
 - MongoDB 是一个介于关系型数据库和非关系型数据中间的产品！MongoDB 是非关系型数据库中功能最丰富，最像关系型数据库的！
- ComtHDB

列存储数据库

I

- HBase
- 分布式文件系统

图关系数据库

Redis入门

概述

<https://www.redis.net.cn/>

什么是 Redis?

Redis (Remote Dictionary Server)，即远程字典服务，是一个开源的使用ANSI C语言编写、支持网络、可基于内存亦可持久化的日志型、Key-Value数据库

redis会周期性的把更新的数据写入磁盘或者把修改操作写入追加的记录文件，并且在此基础上实现了master-slave(主从)同步。

是当下最热门的NoSQL技术之一。

Redis的用处？能干嘛？

Redis 是一个高性能的key-value数据库

Redis 能干嘛 ?

- 1、内存存储、持久化，内存中是断电即失、所以说持久化很重要 (rdb、aof)
- 2、效率高，可以用于高速缓存
- 3、发布订阅系统
- 4、地图信息分析
- 5、计时器、计数器 (浏览量 !)
- 6、

Redis 的特性

特性

- 1、多样的数据类型
- 2、持久化
- 3、集群
- 4、事务

.....

Redis 是单线程的 !

明白 Redis 是很快的，官方表示，Redis 是基于内存操作，CPU 不是 Redis 性能瓶颈，Redis 的瓶颈是根据机器的内存和网络带宽，既然可以使用单线程来实现，就使用单线程了！

Redis 为什么单线程还这么快？

- 1.redis 是基于内存的，内存的读写速度非常快；
- 2.redis 是单线程的，省去了很多上下文切换线程的时间；
- 3.redis 使用多路复用技术，可以处理并发的连接；

Redis 为什么单线程还这么快？

- 1、误区1：高性能的服务器一定是多线程的？
- 2、误区2：多线程 (CPU 上下文会切换！) 一定比单线程效率高！

先去 CPU > 内存 > 硬盘的速度要有所了解！

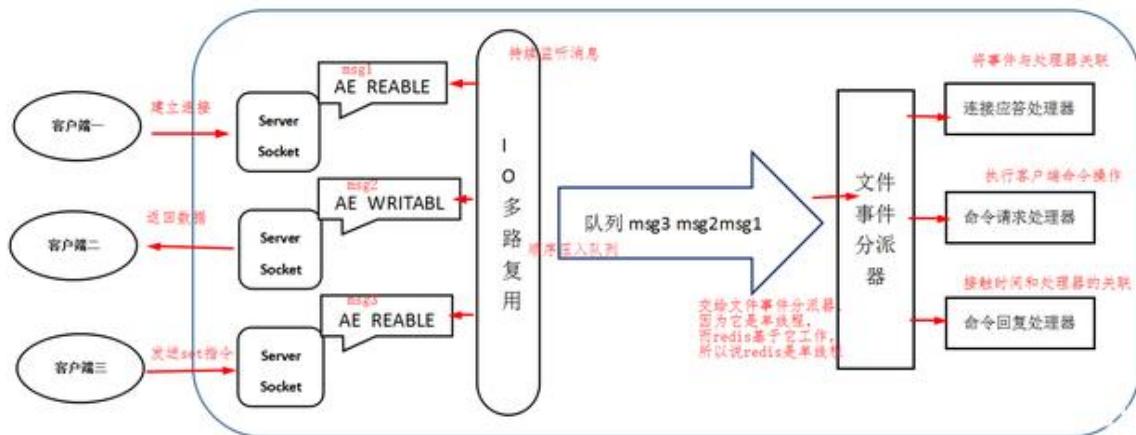
核心：redis 是将所有的数据全部放在内存中的，所以说使用单线程去操作效率就是最高的，多线程 (CPU 上下文会切换：耗时的操作！！！)，对于内存系统来说，如果没有上下文切换效率就是最高的！多次读写都是在一个 CPU 上的，在内存情况下，这个就是最佳的方案！

Redis 单线程到底指什么？

(redis 4 就支持多线程异步删除, redis 6 多线程默认关闭, 可以开启)

Redis 确实是单线程模型, 指的是执行 Redis 命令的核心模块是单线程的, 而不是整个 Redis 实例就一个线程, Redis 其他模块还有各自模块的线程的。

下面这个解释比较好:



Redis基于Reactor模式开发了网络事件处理器，这个处理器被称为文件事件处理器。它的组成结构为4部分：多个套接字、IO多路复用程序、文件事件分派器、事件处理器。因为文件事件分派器队列的消费是单线程的，所以Redis才叫单线程模型。

Docker安装redis

1.下载镜像文件

```
docker pull redis
```

2.创建实例并启动

创建空配置文件

```
mkdir -p /mydata/redis/conf
```

```
touch /mydata/redis/conf/redis.conf
```

创建实例并启动

```
docker run -p 6379:6379 --name redis -v /mydata/redis/data:/data \
-v /mydata/redis/conf/redis.conf:/etc/redis/redis.conf \
-d redis redis-server /etc/redis/redis.conf
```

3.使用 redis 镜像执行 redis-cli

```
docker exec -it redis redis-cli
```

4.redis持久化 (AOF)--现在安装的redis已经默认持久化

```
echo "appendonly yes" >> /mydata/redis/conf/redis.conf  
  
# 重启生效  
docker restart redis
```

容器自启动

```
# redis  
docker update redis --restart=always
```

五大数据类型

Redis 是一个开源 (BSD 许可) 的，内存中的数据结构存储系统，它可以用作数据库、缓存和消息中间件。它支持多种类型的数据结构，如字符串 (strings)，散列 (hashes)，列表 (lists)，集合 (sets)，有序集合 (sorted sets) 与范围查询，bitmaps，hyperloglogs 和地理空间 (geospatial) 索引半径查询。Redis 内置了复制 (replication)，LUA脚本 (Lua scripting)，LRU驱动事件 (LRU eviction)，事务 (transactions) 和不同级别的磁盘持久化 (persistence)，并通过 Redis哨兵 (Sentinel) 和自动分区 (Cluster) 提供高可用性 (high availability)。

关于Redis-Key的基本命令

dokcer 进入redis

```
docker exec -it redis /bin/bash  
redis-cli
```

基本命令

```
127.0.0.1:6379> flushall # 全部清空  
OK  
127.0.0.1:6379> keys * # 查看所有的key  
(empty array)  
127.0.0.1:6379> set name kuangshen # 设置 k-v  
OK  
127.0.0.1:6379> get name # 获取 k-v  
"kuangshen"  
127.0.0.1:6379> flushall  
OK  
127.0.0.1:6379> set name kuangshen  
OK  
127.0.0.1:6379> set age 18  
OK  
127.0.0.1:6379> keys *  
1) "name"  
2) "age"  
127.0.0.1:6379> exists name # 判断指定key是否存在  
(integer) 1  
127.0.0.1:6379> exists name1  
(integer) 0  
127.0.0.1:6379> move name 1 # 将当前数据库的 key 移动到给定的数据库 db 当中
```

```
(integer) 1
127.0.0.1:6379> set name qinjiang
OK
127.0.0.1:6379> keys *
1) "name"
2) "age"
127.0.0.1:6379> expire name 10 # 为给定 key 设置过期时间
(integer) 1
127.0.0.1:6379> ttl name # 以秒为单位, 返回给定 key 的剩余生存时间(TTL, time to live)
(integer) 8
127.0.0.1:6379> ttl name
(integer) 5
127.0.0.1:6379> ttl name
(integer) 1
127.0.0.1:6379> ttl name
(integer) -2
127.0.0.1:6379>
```

String

```
127.0.0.1:6379> set name kuangshen # 设置 k-v
OK
127.0.0.1:6379> get name # 获取 k-v
"kuangshen"
127.0.0.1:6379> flushall
OK
127.0.0.1:6379> set name kuangshen
OK
127.0.0.1:6379> set age 18
OK
127.0.0.1:6379> keys *
1) "name"
2) "age"
127.0.0.1:6379> exists name # 判断指定key是否存在
(integer) 1
127.0.0.1:6379> set name kuangshen
OK
127.0.0.1:6379> append name hello # 如果 key 已经存在并且是一个字符串, APPEND 命令将
value 追加到 key 原来的值的末尾
(integer) 14
127.0.0.1:6379> strlen name # 返回 key 所储存的字符串值的长度
(integer) 14
#####
127.0.0.1:6379> set views 0 # 设置初始浏览量为0
OK
127.0.0.1:6379> get views
"0"
127.0.0.1:6379> incr views # 自增1
(integer) 1
127.0.0.1:6379> incr views
(integer) 2
127.0.0.1:6379> get views
```

```
"2"
127.0.0.1:6379> decr views # 自减1
(integer) 1
127.0.0.1:6379> get views
"1"
127.0.0.1:6379> incrby views 10 # 将 key 所储存的值加上给定的增量值 (increment)
(integer) 11
127.0.0.1:6379> get views
"11"
127.0.0.1:6379> decrby views 5 # key 所储存的值减去给定的减量值 (decrement)
(integer) 7
#####
###
# 字符串范围 range
127.0.0.1:6379> set key1 hello,kuangshen!
OK
127.0.0.1:6379> get key1
"hello,kuangshen!"
127.0.0.1:6379> getrange key1 0 3 # 返回 key 中字符串值的子字符串
"hell"
127.0.0.1:6379> getrange key1 0 -1
"hello,kuangshen!"
127.0.0.1:6379> set key2 abcdefg
OK
127.0.0.1:6379> get key2
"abcdefg"
127.0.0.1:6379> setrange key2 1 xx # 用 value 参数覆写给定 key 所储存的字符串值, 从偏移量 offset 开始
(integer) 7
127.0.0.1:6379> get key2
"axxdefg"
#####
# setex(set with expire) 将值 value 关联到 key , 并将 key 的过期时间设为 seconds (以秒为单位)
# setnx(set if not exist) 如果 key 不存在 就设置 key 的值, 成功返回1, 失败返回0
(在分布式锁中经常使用)

127.0.0.1:6379> setex key3 30 hello # 将值 value 关联到 key , 并将 key 的过期时间设为 seconds
OK
127.0.0.1:6379> ttl key3
(integer) 26
127.0.0.1:6379> ttl key3
(integer) 22
127.0.0.1:6379> setnx mykey redisyyds! # 如果 key 不存在 就设置 key 的值
(integer) 1
127.0.0.1:6379> keys *
1) "key2"
2) "key1"
3) "mykey"
127.0.0.1:6379> setnx mykey "redis is yyds"
(integer) 0
127.0.0.1:6379>
#####
# mset
# mget
```

```
# msetnx 原子性操作，要么全部成功，要么全部失败
```

```
127.0.0.1:6379> mset key1 v1 key2 v2 key3 v3 # 同时设置一个或多个 key-value 对
OK
127.0.0.1:6379> keys *
1) "key3"
2) "key2"
3) "key1"
127.0.0.1:6379> mget key1 key2 key3 # 获取所有(一个或多个)给定 key 的值
1) "v1"
2) "v2"
3) "v3"
127.0.0.1:6379> msetnx key1 v1 key4 v4 # 同时设置一个或多个 key-value 对，当且仅当所有给定 key 都不存在
(integer) 0
#####
# getset (先get再set)将给定 key 的值设为 value，并返回 key 的旧值(old value)
127.0.0.1:6379> getset db redis
(nil)
127.0.0.1:6379> get db
"redis"
127.0.0.1:6379> getset db mongodb
"redis"
127.0.0.1:6379> get db
"mongodb"

# 对象
127.0.0.1:6379> mset user1:1:name zhangsan user1:1:age 19
OK
127.0.0.1:6379> keys *
1) "db"
2) "user1:1:age"
3) "user1:1:name"
127.0.0.1:6379> mget user1:1:name
1) "zhangsan"
127.0.0.1:6379> mget user1:1:age
1) "19"
127.0.0.1:6379>
```

使用场景

value除了是字符串，还可以是数字

- 计数器
- 统计多单位的数量
- 粉丝数
- 对象缓存存储

List (列表)

双向链表

- list可以用作栈、队列、阻塞队列等等。。。
- 所有的 list 命令都是 l 开头的

```
#####
```

```
# lpush
# rpush
# lrange
127.0.0.1:6379> lpush list one # 将一个或多个值插入到列表头部
(integer) 1
127.0.0.1:6379> lpush list two
(integer) 2
127.0.0.1:6379> lpush list three
(integer) 3
127.0.0.1:6379> lrange list 0 -1 # 获取列表指定范围内的元素
1) "three"
2) "two"
3) "one"
127.0.0.1:6379> lrange list 0 1
1) "three"
2) "two"
127.0.0.1:6379> rpush list four # 在列表尾部添加一个或多个值
(integer) 4
127.0.0.1:6379> lrange list 0 -1
1) "three"
2) "two"
3) "one"
4) "four"
127.0.0.1:6379>
#####
# lpop
# rpop
127.0.0.1:6379> keys *
(empty array)
127.0.0.1:6379> lpush list one two three
(integer) 3
127.0.0.1:6379> rpush list four
(integer) 4
127.0.0.1:6379> lrange list 0 -1
1) "three"
2) "two"
3) "one"
4) "four"
127.0.0.1:6379> lpop list # 移除头节点
"three"
127.0.0.1:6379> rpop list # 移除尾节点
"four"
127.0.0.1:6379> lrange list 0 -1
1) "two"
2) "one"
#####
# lindex
127.0.0.1:6379> lrange list 0 -1
1) "two"
2) "one"
127.0.0.1:6379> lindex list 1 # 获取列表中指定下标的元素（可以用作阻塞队列，生产者消费者模式）
"one"
127.0.0.1:6379> lindex list 0
"two"
#####
# llen
127.0.0.1:6379> llen list # 获取指定列表的长度
```

```
(integer) 2
#####
# lrem 移除list集合中指定个数的value
127.0.0.1:6379> flushall
OK
127.0.0.1:6379> lpush list one
(integer) 1
127.0.0.1:6379> lpush list two
(integer) 2
127.0.0.1:6379> lpush list three
(integer) 3
127.0.0.1:6379> lpush list three
(integer) 4
127.0.0.1:6379> lrange list 0 -1
1) "three"
2) "three"
3) "two"
4) "one"
127.0.0.1:6379> lrem list 1 one
(integer) 1
127.0.0.1:6379> lrange list 0 -1
1) "three"
2) "three"
3) "two"
127.0.0.1:6379> lrem list 2 three
(integer) 2
127.0.0.1:6379> lrange list 0 -1
1) "two"
#####
# ltrim 对一个列表进行修剪(trim)，就是说，让列表只保留指定区间内的元素，不在指定区间之内的元素都将被删除
127.0.0.1:6379> flushdb
OK
127.0.0.1:6379> keys *
(empty array)
127.0.0.1:6379> lpush mylist hello
(integer) 1
127.0.0.1:6379> lpush mylist hello1
(integer) 2
127.0.0.1:6379> lpush mylist hello2
(integer) 3
127.0.0.1:6379> lpush mylist hello3
(integer) 4
127.0.0.1:6379> lrange 0 -1
(error) ERR wrong number of arguments for 'lrange' command
127.0.0.1:6379> lrange mylist 0 -1
1) "hello3"
2) "hello2"
3) "hello1"
4) "hello"
127.0.0.1:6379> ltrim mylist 1 2
OK
127.0.0.1:6379> lrange mylist 0 -1
1) "hello2"
2) "hello1"
#####
# rpoplpush 移除列表的第一个元素，并将该元素添加到另一个列表并返回
```

```
127.0.0.1:6379> rpush mylist hello
(integer) 1
127.0.0.1:6379> rpush mylist hello1
(integer) 2
127.0.0.1:6379> rpush mylist hello2
(integer) 3
127.0.0.1:6379> rpush mylist hello3
(integer) 4
127.0.0.1:6379> rpoplpush mylist myotherlist
"hello3"
127.0.0.1:6379> lrange mylist 0 -1
1) "hello"
2) "hello1"
3) "hello2"
127.0.0.1:6379> lrange myotherlist 0 -1
1) "hello3"
#####
# exists 检查给定 key 是否存在
# lset 通过索引设置列表元素的值
127.0.0.1:6379> exists list # exists 检查给定 key 是否存在
(integer) 0
127.0.0.1:6379> exists mylist
(integer) 1
127.0.0.1:6379> exists myotherlist
(integer) 1
127.0.0.1:6379> lpush list value1
(integer) 1
127.0.0.1:6379> lpush list value2
(integer) 2
127.0.0.1:6379> lrange list 0 -1
1) "value2"
2) "value1"
127.0.0.1:6379> lset list 0 valuevalue # lset 通过索引更新列表元素的值, 如果不存在就报错
OK
127.0.0.1:6379> lrange list 0 -1
1) "valuevalue"
2) "value1"
127.0.0.1:6379> lset list 9 valuevalue
(error) ERR index out of range
#####
# linsert 在列表的元素前或者后插入元素
127.0.0.1:6379> lpush list value
(integer) 1
127.0.0.1:6379> lpush list value1
(integer) 2
127.0.0.1:6379> lpush list value2
(integer) 3
127.0.0.1:6379> lpush list value3
(integer) 4
127.0.0.1:6379> linsert list before value2 hello
(integer) 5
127.0.0.1:6379> lrange list 0 -1
1) "value3"
2) "hello"
3) "value2"
4) "value1"
5) "value"
127.0.0.1:6379> linsert list after value2 word
```

```
(integer) 6
127.0.0.1:6379> lrange list 0 -1
1) "value3"
2) "hello"
3) "value2"
4) "word"
5) "value1"
6) "value"

#####
#
```

请记住，只有push和pop时才分左右，其余的命令的L均为list的意思

小结

- List 实际是一个链表
- List中的值可以重复

应用

- 消息队列

Set (集合)

```
# sadd 向集合添加一个或多个成员
# smembers 返回集合中的所有成员
# sismember 判断 member 元素是否是集合 key 的成员
127.0.0.1:6379> sadd myset hello
(integer) 1
127.0.0.1:6379> sadd myset kuangshen
(integer) 1
127.0.0.1:6379> smembers myset
1) "hello"
2) "kuangshen"
127.0.0.1:6379> sismember myset hello
(integer) 1
127.0.0.1:6379> sismember myset niubi
(integer) 0
#####
#scard 获取集合的成员数
127.0.0.1:6379> scard myset
(integer) 2
#####
# srem 移除集合中一个或多个成员
127.0.0.1:6379> sadd myset newvalue
(integer) 1
127.0.0.1:6379> smembers myset
1) "hello"
2) "kuangshen"
3) "newvalue"
127.0.0.1:6379> srem myset hello
(integer) 1
127.0.0.1:6379> smembers myset
1) "kuangshen"
2) "newvalue"
#####
# srandom 返回集合中一个或多个随机数
#
```

```
127.0.0.1:6379> srandmember myset
#####
# spop 移除并返回集合中指定个数的随机元素

127.0.0.1:6379> spop myset
#####
# smove 将 member 元素从 source 集合移动到 destination 集合
127.0.0.1:6379> sadd myset hello
(integer) 1
127.0.0.1:6379> sadd myset vlaue
(integer) 1
127.0.0.1:6379> sadd myset word
(integer) 1
127.0.0.1:6379> smembers myset
1) "word"
2) "hello"
3) "vlaue"
127.0.0.1:6379> smove myset myotherset hello
(integer) 1
127.0.0.1:6379> smembers myset
1) "word"
2) "vlaue"
127.0.0.1:6379> smembers myotherset
1) "hello"

#####
# 微博、B站的共同关注列表
# sdiff 返回给定所有集合的差集
# sinter 返回给定所有集合的交集
# sunion 返回所有给定集合的并集
127.0.0.1:6379> sadd myset1 a b c
(integer) 3
127.0.0.1:6379> sadd myset2 c d e
(integer) 3
127.0.0.1:6379> smembers myset1
1) "c"
2) "b"
3) "a"
127.0.0.1:6379> smembers myset2
1) "d"
2) "c"
3) "e"
127.0.0.1:6379> sdiff myset1 myset2
1) "b"
2) "a"
127.0.0.1:6379> sunion myset1 myset2
1) "b"
2) "c"
3) "a"
4) "d"
5) "e"
127.0.0.1:6379> sinter myset1 myset2
1) "c"

#####
```

小结

- Set中的值不可以重复

应用

- 微博、B站的共同关注列表
- 推荐好友（六度分隔理论？）

Hash (哈希)

Map集合，key-map

```
#####
127.0.0.1:6379> hset myhash field1 kuangshen # hset 将哈希表 key 中的字段 field 的值
设为 value
(integer) 1
127.0.0.1:6379> hget myhash field1 # hget 获取存储在哈希表中指定字段的值
"kuangshen"
127.0.0.1:6379> hset myhash field2 hello field3 word
(integer) 2
127.0.0.1:6379> hget myhash field3
"word"

127.0.0.1:6379> hgetall myhash # hgetall 获取在哈希表中指定 key 的所有字段和值
1) "field1"
2) "kuangshen"
3) "field2"
4) "hello"
5) "field3"
6) "word"
#####
# hdel 删除一个或多个哈希表字段
127.0.0.1:6379> hdel myhash field1 field2
(integer) 2

#####
# hlen 获取哈希表中字段(键值对)的数量
127.0.0.1:6379> hlen myhash
(integer) 1

#####
# hexists 查看哈希表 key 中, 指定的字段是否存在
127.0.0.1:6379> hexists myhash field1
(integer) 0

#####
# hkeys 获取所有哈希表中的字段
# hvals 获取哈希表中所有值
127.0.0.1:6379> hgetall myhash
1) "field3"
2) "word"
3) "field1"
4) "hello"
5) "field2"
6) "kuangshen"
```

```

127.0.0.1:6379> hvals myhash
1) "word"
2) "hello"
3) "kuangshen"
127.0.0.1:6379> hkeys myhash
1) "field3"
2) "field1"
3) "field2"
#####
# hincrby 为哈希表 key 中的指定字段的整数值加上增量 increment
127.0.0.1:6379> hset myhash field4 4
(integer) 1
127.0.0.1:6379> hincrby myhash field4 3
(integer) 7
127.0.0.1:6379> hincrby myhash field4 -3
(integer) 4

#####
# hsetnx 只有在字段 field 不存在时，设置哈希表字段的值
127.0.0.1:6379> hsetnx myhash field4 99
(integer) 0
127.0.0.1:6379> hsetnx myhash field5 99
(integer) 1

#####

```

总结

- Map集合, key-map
- 根据Redis 4.0.0, HMSET被视为已弃用。请在新代码中使用HSET

redis的hash与string区别

Redis hash 是一个 string 类型的 field 和 value 的映射表，它的添加、删除操作都是 **O(1)** (平均操作)。

hash 特别适合用于存储对象。相较于将对象的每个字段存成单个 string 类型 (string 类型可以存储对象序列化)。

将一个对象存储在 hash 类型中会占用更少的内存，并且可以更方便的存取整个对象。（省内存的原因是新建一个 hash 对象时开始是用 zipmap (又称为 small hash) 来存储的。

这个 zipmap 其实并不是 hash table，但是 zipmap 相比正常的 hash 实现可以节省不少 hash 本身需要的一些元数据存储开销。

Zset (有序集合)

```

#####
127.0.0.1:6379> zadd myset 1 one # zadd 向有序集合添加一个或多个成员，或者更新已存在成员
的分数
(integer) 1
127.0.0.1:6379> zadd myset 2 two 3 three
(integer) 2
127.0.0.1:6379> zrange myset 2 3
1) "three"

#####

```

```
127.0.0.1:6379> zrange myset 0 -1 # zrange 通过索引区间返回有序集合成指定区间内的成员
1) "one"
2) "two"
3) "three"
127.0.0.1:6379> zadd salary 2500 xiaohong 5000 zhangsan
(integer) 2
127.0.0.1:6379> zadd salary 500 kuangshen
(integer) 1
127.0.0.1:6379> zrange salary 0 -1
1) "kuangshen"
2) "xiaohong"
3) "zhangsan"
```

```
#####
127.0.0.1:6379> zrangebyscore salary -inf +inf
# zrangebyscore 通过分数返回有序集合指定区间内的成员，从小到大排序
# zrangebyscore 返回有序集中指定分数区间内的成员，分数从高到低排序
# -inf 负无穷，+inf 正无穷
1) "kuangshen"
2) "xiaohong"
3) "zhangsan"
127.0.0.1:6379> zrangebyscore salary -inf +inf withscores
1) "kuangshen"
2) "500"
3) "xiaohong"
4) "2500"
5) "zhangsan"
6) "5000"
127.0.0.1:6379>
```

```
#####
# zrem 移除有序集合中的一个或多个成员
127.0.0.1:6379> zrange salary 0 -1
1) "kuangshen"
2) "xiaohong"
3) "zhangsan"
127.0.0.1:6379> zrem salary xiaohong
(integer) 1
127.0.0.1:6379> zrange salary 0 -1
1) "kuangshen"
2) "zhangsan"
```

```
#####
# zcount 计算在有序集合中指定区间分数的成员数
127.0.0.1:6379> zadd myset 1 one
(integer) 1
127.0.0.1:6379> zadd myset 2 two
(integer) 1
127.0.0.1:6379> zadd myset 3 three
(integer) 1
127.0.0.1:6379> zcount myset 1 3
(integer) 3
127.0.0.1:6379> zcount myset 2 3
(integer) 2
```

总结

- 在 set 的基础上增加了一个值
- 底层跳表实现

应用

- 存储班级成绩
- 工资表
- 带权重的消息存储
- 排行榜 (例: B 站排行榜)

三种特殊数据类型

Geospatial 地理位置详解

```
# geoadd 将指定的地理空间位置（纬度、经度、名称）添加到指定的key中
127.0.0.1:6379> geoadd china:city 117.092 40.628 beijing
(integer) 1
127.0.0.1:6379> geoadd china:city 113.273 23.157 guangzhou
(integer) 1
127.0.0.1:6379> geoadd china:city 106.540 29.402 chongqing
(integer) 1
127.0.0.1:6379> geoadd china:city 122.187 29.902 shanghai
(integer) 1
#####
# geopos 从key里返回所有给定位置元素的位置（经度和纬度）
127.0.0.1:6379> geopos china:city beijing
1) 1) "117.09199815988540649"
   2) "40.62799989056582461"
127.0.0.1:6379> geopos china:city guangzhou shanghai
1) 1) "113.27299922704696655"
   2) "23.15700086250385681"
2) 1) "122.18699902296066284"
   2) "29.90200032579349454"
#####
# geodist 返回两个给定位置之间的距离
127.0.0.1:6379> geodist china:city guangzhou shanghai km
"1160.9585"

#####
# georadius 以给定的经纬度为中心， 找出某一半径内的元素
127.0.0.1:6379> georadius china:city 122 29 1000 km
1) "shanghai"
127.0.0.1:6379> georadius china:city 122 29 10000 km
1) "chongqing"
2) "guangzhou"
3) "shanghai"
4) "beijing"

#####
# georadiusbymember 找出位于指定范围内的元素，中心点是由给定的位置元素决定
127.0.0.1:6379> georadiusbymember china:city shanghai 1500 km
1) "guangzhou"
2) "shanghai"
```

```
3) "beijing"
#####
# geohash 返回一个或多个位置元素的 Geohash 表示
127.0.0.1:6379> geohash china:city shanghai
1) "wtqfh7mzg20"
```

总结

- geo底层实现：zset， 可以使用zset命令操作geo

- 127.0.0.1:6379> zrange china:city 0 -1
1) "chongqing"
2) "guangzhou"
3) "shanghai"
4) "beijing"
127.0.0.1:6379> zrem china:city chongqing
(integer) 1
127.0.0.1:6379> zrange china:city 0 -1
1) "guangzhou"
2) "shanghai"
3) "beijing"

应用

- 朋友定位
- 附近的人
- 打车距离计算

经纬度查询：<https://jingweidu.bmcx.com/>

Hyperloglog 基数统计

```
127.0.0.1:6379> pfadd mykey1 a b c d e f g
(integer) 1
127.0.0.1:6379> pfcount mykey1
(integer) 7
127.0.0.1:6379> pfadd mykey2 e f g h i j k l m n o
(integer) 1
127.0.0.1:6379> pfcount mykey2
(integer) 11
127.0.0.1:6379> pfmerge mykey3 mykey1 mykey2
OK
127.0.0.1:6379> pfcount mykey3
(integer) 15
```

Redis **HyperLogLog** 是用来做基数统计的算法，HyperLogLog 的优点是，在输入元素的数量或者体积非常非常大时，计算基数所需的空间总是固定 的、并且是很小的，**HLL有一定的错误率**

但是，因为 HyperLogLog 只会根据输入元素来计算基数，而不会储存输入元素本身，所以 HyperLogLog 不能像集合那样，返回输入的各个元素。

什么是基数?

比如数据集 {1, 3, 5, 7, 5, 7, 8}, 那么这个数据集的基数集为 {1, 3, 5, 7, 8}, 基数(不重复元素)为5。基数估计就是在误差可接受的范围内, 快速计算基数。

HyperLogLog与布隆过滤器

HyperLogLog与布隆过滤器都是针对大数据统计存储应用场景下的知名算法。

HyperLogLog是在大数据的情况下关于数据基数的空间复杂度优化实现, 布隆过滤器是在大数据情况下关于检索一个元素是否在一个集合中的空间复杂度优化后的实现。

应用

- 网站浏览量统计

如果允许容错使用 HLL, 如果不允许容错就使用 set或自定义的数据类型

Bitmap 位图场景详解

实例:

使用bitmap 记录周一到周日的打卡

```
# setbit 设置
127.0.0.1:6379> setbit sign 1 1
(integer) 0
127.0.0.1:6379> setbit sign 2 1
(integer) 0
127.0.0.1:6379> setbit sign 3 1
(integer) 0
127.0.0.1:6379> setbit sign 4 0
(integer) 0
127.0.0.1:6379> setbit sign 5 0
(integer) 0
127.0.0.1:6379> setbit sign 6 1
(integer) 0
127.0.0.1:6379> setbit sign 7 0
(integer) 0

#####
# getbit 获取
127.0.0.1:6379> getbit sign 1
(integer) 1
127.0.0.1:6379> getbit sign 6
(integer) 1
127.0.0.1:6379> getbit sign 5
(integer) 0

#####
# bitcount 统计打卡天数
127.0.0.1:6379> bitcount sign
(integer) 4
```

应用

两种状态的都可以使用 位图

- 统计用户信息：活跃、不活跃
- 登录与未登录
- 打卡

redis基本的事务操作

redis单条命令是保证原子性的，但是redis的事务不保证原子性！

redis事务三个特性

- 顺序性
- 一次性
- 排他性

Redis 事务可以一次执行多个命令，并且带有以下三个重要的保证：

- 批量操作在发送 EXEC 命令前被放入队列缓存
- 收到 EXEC 命令后进入事务执行，事务中任意命令执行失败，其余的命令依然被执行
- 在事务执行过程，其他客户端提交的命令请求不会插入到事务执行命令序列中

一个事务从开始到执行会经历以下三个阶段：

- 开始事务
- 命令入队
- 执行事务

正常执行事务

```
127.0.0.1:6379> multi # 开启事务
OK
# 命令入队
127.0.0.1:6379(TX)> set k1 v1
QUEUED
127.0.0.1:6379(TX)> set k2 v2
QUEUED
127.0.0.1:6379(TX)> get k2
QUEUED
127.0.0.1:6379(TX)> set k3 v3
QUEUED
# 执行事务
127.0.0.1:6379(TX)> exec
1) OK
```

```
2) OK
3) "v2"
4) OK
```

放弃事务

一旦放弃事务，事务中的所有命令都不会被执行

```
# 开启事务
127.0.0.1:6379> multi
OK
# 命令入队
127.0.0.1:6379(TX)> set k3 v3
QUEUED
127.0.0.1:6379(TX)> set k4 v4
QUEUED
# 放弃事务
127.0.0.1:6379(TX)> discard
OK
127.0.0.1:6379> get k4
(nil)
```

编译型异常

代码写错了（命令写错了），事务中的所有命令都不会被执行

```
127.0.0.1:6379> multi
OK
127.0.0.1:6379(TX)> set k1 v1
QUEUED
127.0.0.1:6379(TX)> set k2 v2
QUEUED
127.0.0.1:6379(TX)> set k3 v3
QUEUED
127.0.0.1:6379(TX)> set k9 v9
QUEUED
# -----
127.0.0.1:6379(TX)> getset k9 # 错误的命令
(error) ERR wrong number of arguments for 'getset' command
127.0.0.1:6379(TX)> exec
(error) EXECABORT Transaction discarded because of previous errors.
127.0.0.1:6379> get k9
(nil)
```

运行时异常

事务队列中存在语法性错误，那么执行事务的时候其他命令会正常执行，错误命令会抛出异常

```
127.0.0.1:6379> multi
OK
127.0.0.1:6379(TX)> set k1 "abc"
QUEUED
127.0.0.1:6379(TX)> incr k1
QUEUED
127.0.0.1:6379(TX)> set k2 v2
QUEUED
127.0.0.1:6379(TX)> exec
1) OK
2) (error) ERR value is not an integer or out of range
3) OK
```

redis 实现乐观锁 (watch)

监控！ Watch

悲观锁：

- 很悲观，认为什么时候都会出问题，无论做什么都会加锁！

乐观锁：

- 很乐观，认为什么时候都不会出问题，所以不会上锁！更新数据的时候去判断一下，在此期间是否有人修改过这个数据，
- 获取version
- 更新的时候比较version

无论事务是否执行成功，Redis都会取消Watch监控

redis对事务的支持比较简单。redis只能保证一个客户端发起的事务命令可以执行，中间不会插入其他事务。因为redis是单线程的，所以做到上面这点很容易。一般redis接受到客户端的命令后会立即执行，但是如果客户端发起multi命令，redis不会立即执行，而是让当前连接进入事务上下文，把命令放到队列中，接受到exec命令后，redis会顺序执行队列中的命令。并把执行结果打包到一起返回客户端，之后就结束了事务上下文

实例

当前线程在执行事务时，其他线程修改了值，那么当前线程就会执行失败

```
127.0.0.1:6379> set money 100
OK
127.0.0.1:6379> set out 0
OK
127.0.0.1:6379> watch money # 监控（加锁）
OK
#####
# 当前线程开启事务
127.0.0.1:6379> multi
OK
127.0.0.1:6379(TX)> decrby money 20
QUEUED
```

```
127.0.0.1:6379(TX)> incrby out 20
QUEUED
#####
# 其他线程修改 money
127.0.0.1:6379> get money
"100"
127.0.0.1:6379> incrby money 1000
(integer) 1100

#####
# 当前线程执行事务
127.0.0.1:6379(TX)> exec
(nil) # 执行失败
```

redis是单线程的，客户端发起multi命令，redis不会立即执行

总结

- 无论事务是否执行成功，Redis都会取消Watch监控
- unwatch 解开全部 watch，执行exec或者discard就不用执行unwatch了
- 如果修改失败，重新监控执行就好

Jredis 操作 redis

Jedis

我们要使用 Java 来操作 Redis

什么是Jedis 是 Redis 官方推荐的 java连接开发工具！ 使用java 操作Redis 中间件！如果你要使用 java操作redis，那么一定要对jedis十分的熟悉！

SpringBoot 集成redis

SpringData Redis 底层使用： [Lettuce](#) and [Jedis](#)

Redis 配置

redis启动时候通过配置文件来启动

1、单位

redis配置文件对大小写不敏感

```
# Redis configuration file example

# Note on units: when memory size is needed, it is possible to specify
# it in the usual form of 1k 5GB 4M and so forth:
#
# 1k => 1000 bytes
# 1kb => 1024 bytes
# 1m => 1000000 bytes
# 1mb => 1024*1024 bytes
# 1g => 1000000000 bytes
# 1gb => 1024*1024*1024 bytes
#
# units are case insensitive so 1GB 1Gb 1gB are all the same.

#####
##### INCLUDES #####
#####
```

2、可包含其他配置文件

```
#####
##### INCLUDES #####
#
# Include one or more other config files here. This is useful if you
# have a standard template that goes to all Redis servers but also need
# to customize a few per-server settings. Include files can include
# other files, so use this wisely.
#
# Notice option "include" won't be rewritten by command "CONFIG REWRITE"
# from admin or Redis Sentinel. Since Redis always uses the last processed
# line as value of a configuration directive, you'd better put includes
# at the beginning of this file to avoid overwriting config change at runtime.
#
# If instead you are interested in using includes to override configuration
# options, it is better to use include as the last line.
#
# include .\path\to\local.conf
# include c:\path\to\other.conf
#
```

3、网络

```
bind 127.0.0.1 # 绑定的ip
protected-mode yes # 保护模式 默认开启
port 6379 # 端口, 默认6379
```

4、通用配置

```
daemonize no # 以守护进程方式运行， 默认为no
pidfile /var/run/redis.pid # 如果以后台方式运行， 就需要指定 pid文件

# Specify the server verbosity level.
# This can be one of:
# debug (a lot of information, useful for development/testing)
# verbose (many rarely useful info, but not a mess like the debug level)
# notice (moderately verbose, what you want in production probably)
# warning (only very important / critical messages are logged)
loglevel notice # 日志级别
logfile "server_log.txt" # 日志文件名
databases 16 # 数据库数量， 默认16
always-show-logo yes # 是否显示logo
```

5、快照

持久化：在规定的时间内执行了多少次操作就进行持久化到文件（.rdb或.aof）

redis 是内存数据库，如果不进行持久化那就会断电即失

```
save 900 1 # 若 900秒内进行了一次操作，则进行持久化
save 300 10
save 60 10000

stop-writes-on-bgsave-error yes # 当bgsave快照操作出错时停止写数据到磁盘
rdbcompression yes # 压缩rdb文件（会消耗一些cpu资源）
rdbchecksum yes # rdb文件校验
dir ./ # rdb文件的保存目录
```

6、主从复制 REPLICATION

7、安全 SECURITY

```
requirepass "" # 设置密码
```

8、客户端 CLIENTS

```
maxclients 10000 # 最大客户端连接数
```

9、内存管理 MEMORY MANAGEMENT

```
maxmemory <bytes> # redis配置最大内存容量
maxmemory-policy noevasion # redis内存管理策略
```

redis在占用的内存超过指定的maxmemory之后，通过maxmemory_policy确定redis是否释放内存以及如何释放内存。redis提供了**8种内存超过限制之后的响应措施**，分别如下：

1. **volatile-lru**(least recently used): 最近最少使用算法, 从设置了过期时间的键中选择空转时间最长的键值对清除掉;
2. **volatile-lfu**(least frequently used): 最近最不经常使用算法, 从设置了过期时间的键中选择某段时间之内使用频次最小的键值对清除掉;
3. **volatile-ttl**: 从设置了过期时间的键中选择过期时间最早的键值对清除;
4. **volatile-random**: 从设置了过期时间的键中, 随机选择键进行清除;
5. **allkeys-lru**: 最近最少使用算法, 从所有的键中选择空转时间最长的键值对清除;
6. **allkeys-lfu**: 最近最不经常使用算法, 从所有的键中选择某段时间之内使用频次最少的键值对清除;
7. **allkeys-random**: 所有的键中, 随机选择键进行删除;
8. **noeviction**: 不做任何的清理工作, 在redis的内存超过限制之后, 所有的写入操作都会返回错误; 但是读操作都能正常的进行;

10、AOF配置 APPEND ONLY MODE

```
appendonly no # 默认不开启AOF
appendfilename "appendonly.aof" # AOF持久化文件

# appendfsync always # 每次修改都进行同步
appendfsync everysec # 每秒执行一次同步, 可能会丢失这一秒的数据
# appendfsync no # 不执行同步, 这个时候操作系统自行同步数据,
```

Redis 持久化 (重点)

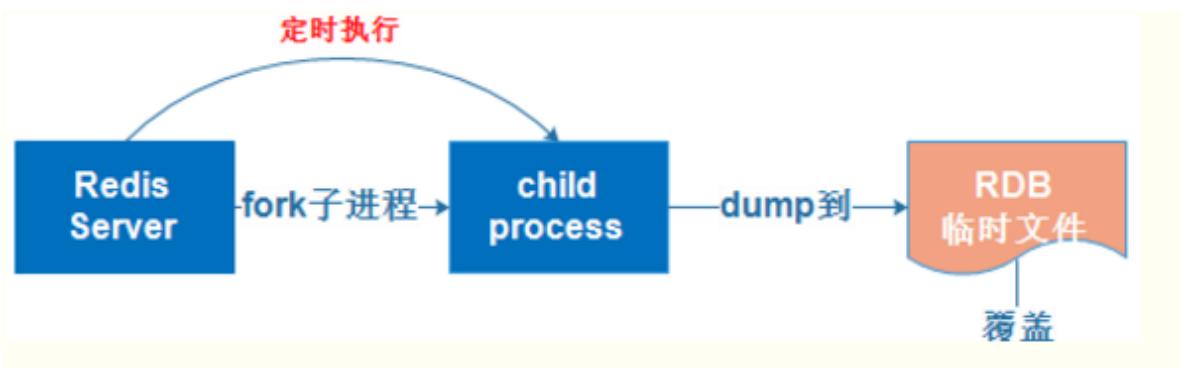
由于Redis的数据都存放在内存中, 如果没有配置持久化, redis重启后数据就全丢失了, 于是需要开启redis的持久化功能, 将数据保存到磁盘上, 当redis重启后, 可以从磁盘中恢复数据。

redis提供两种方式进行持久化, 一种是RDB持久化 (原理是将Reids在内存中的数据库记录定时dump到磁盘上的RDB持久化), 另外一种是AOF (append only file) 持久化 (原理是将Reids的操作日志以追加的方式写入文件)

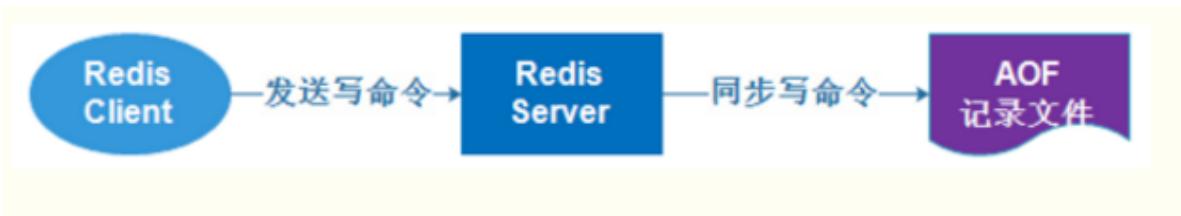
redis还可以同时使用AOF持久化和RDB持久化, 在这种情况下, 当redis重启时, 它会有限使用AOF文件来还原数据集, 因为AOF文件保存的数据集通常比RDB文件所保存的数据集更加完整

二者的区别

RDB持久化是指在指定的时间间隔内将内存中的数据集快照写入磁盘, 实际操作过程是fork一个子进程, 先将数据集写入临时文件, 写入成功后, 再替换之前的文件, 用二进制压缩存储。RDB的缺点是最后一次持久化后的数据可能会丢失。



AOF持久化以日志的形式记录服务器所处理的每一个写、删除操作，查询操作不会记录，只会追加文件但不会改写文件，redis重启后会根据日志文件的内容将指令从头到尾执行一次以完成数据的恢复操作。



RDB

配置

```
save 900 1
save 300 10
save 60 10000
```

优缺点

优点

1. RDB 是一个非常紧凑 (compact) 的文件，它保存了 Redis 在某个时刻上的数据集。这种文件非常适合用于进行**冷备份**：比如说，你可以在最近的 24 小时内，每小时备份一次 RDB 文件，并且在每个月的每一天，也备份一个 RDB 文件。这样的话，即使遇上问题，**也可以随时将数据集还原到不同的版本**。
2. **RDB 非常适用于灾难恢复** (disaster recovery)：它只有一个文件，并且内容都非常紧凑，可以在 (加密后) 将它传送到别的数据中心，或者亚马逊 S3 中。
3. **RDB 可以最大化 Redis 的性能**：父进程在保存 RDB 文件时唯一要做的就是 fork 出一个子进程，然后这个子进程就会处理接下来的所有保存工作，父进程无须执行任何磁盘 I/O 操作。
4. **RDB 在恢复大数据集时的速度比 AOF 的恢复速度要快**

缺点

1. redis 故障时，由于周期性持久化的**存在时间间隔**的问题，可能会**丢失间隔时间内产生的数据**，这一点上 RDB 没有 AOF 好
2. RDB 每次在 fork 子进程来执行 RDB 快照数据文件生成的时候，如果数据文件特别大，可能会导致对客户端提供的服务暂停数毫秒，或者甚至数秒。
 - 每次保存 RDB 的时候，Redis 都要 `fork()` 出一个子进程，并由子进程来进行实际的持久化工作。在数据集比较庞大时，`fork()` 可能会非常耗时，造成服务器在某某毫秒内停止处理

客户端；如果数据集非常巨大，并且 CPU 时间非常紧张的话，那么这种停止时间甚至可能会长达整整一秒。虽然 AOF 重写也需要进行 `fork()`，但无论 AOF 重写的执行间隔有多长，数据的耐久性都不会有任何损失。

RDB持久化机制

在Redis运行时，RDB程序将当前内存中的数据库快照保存到磁盘文件中，在Redis重启时，RDB程序可以通过载入RDB文件来还原数据库的状态。RDB机制最主要的就是`rdbSave`和`rdbLoad`函数，前者将redis内存中数据加载到磁盘上，后者将在Redis重启时将数据恢复到redis内存中，注意`rdbSave`会阻塞主进程。

SAVE命令和BGSAVE命令的区别

`SAVE` 和 `BGSAVE` 两个命令都会调用`rdbSave`函数，但它们调用的方式各有不同：

- 1, `SAVE` 直接调用`rdbSave`，阻塞Redis主进程，直到保存完成为止。在主进程阻塞期间，服务器不能处理客户端的任何请求。
- 2, `BGSAVE` 则`fork`出一个子进程，子进程负责调用`rdbSave`，并在保存完成之后向主进程发送信号，通知保存已完成。因为`rdbSave` 在子进程被调用，所以Redis 服务器在`BGSAVE` 执行期间仍然可以继续处理客户端的请求。

SAVE命令和BGSAVE命令的选择

`SAVE`命令在创建RDB文件期间会阻塞Redis服务器，所以如果我们需要在创建RDB文件的同时让Redis服务器继续为其他客户端服务，那么就只能使用`BGSAVE`命令来创建RDB文件，`bgsave`会创建子进程在后台进行持久化，不会阻塞redis服务

因为`SAVE`命令无须创建子进程，它不会因为创建子进程而消耗额外的内存，所以在维护离线的Redis服务器时，使用`SAVE`命令能够比使用`BGSAVE`命令更快地完成创建RDB文件的工作。

生成RDB文件的触发机制

1. `save` 规则满足，会自动触发
2. 执行 `flushall`，会触发
3. 退出 redis，会触发

`kill` 直接杀死进程，redis无法保存

备份就会生成 .rdb文件

如何恢复 RDB文件？

1. 将 rdb文件放在 redis启动目录
2. 执行命令：`config get dir`,

```
127.0.0.1:6379> config get dir
1) "dir"
2) "/data" # 如果对应目录下存在 rdb文件启动就会自动恢复其中的数据
```

AOF (Append Only File)

AOF 持久化记录服务器执行的所有写操作命令，并在服务器启动时，通过重新执行这些命令来还原数据集。AOF 文件中的命令全部以 Redis 协议的格式来保存，新命令会被追加到文件的末尾。默认文件名：appendonly.aof

配置

```
appendonly no

appendfilename "appendonly.aof"

# appendfsync always
appendfsync everysec
# appendfsync no

auto-aof-rewrite-percentage 100 # 增长百分比为100时开启重写（默认是100）
auto-aof-rewrite-min-size 64mb # 当前aof文件大小大于这个值开启重写(fork一个子进程进行)
```

appendonly.aof数据文件损坏的时候如何修复

先进行备份，然后使用 Redis 附带的 `redis-check-aof` 程序，对原来的 AOF 文件进行修复，进而再启动 redis

```
cp appendonly.aof appendonly.aof.bak
redis-check-aof --fix appendonly.aof
```

AOF 的优缺点

优点

1. **AOF 可以更好的保护数据不丢失**，一般从效率和安全方面综合考虑设置 AOF 每隔 1 秒通过一个后台线程执行一次 fsync 操作，最多丢失 1 秒钟的数据（`fsync` 会在后台线程执行，所以主线程可以继续努力地处理命令请求）。
2. AOF 日志文件以 append-only 模式写入，所以没有任何磁盘寻址的开销，**写入性能非常高**，而且文件不容易破损，即使文件尾部破损，也很容易修复。
 - AOF 文件是一个只进行追加操作的日志文件（append only log），因此对 AOF 文件的写入不需要进行 `seek`，即使日志因为某些原因而包含了未写入完整的命令（比如写入时磁盘已满，写入中途停机，等等），`redis-check-aof` 工具也可以轻易地修复这种问题。
3. **Redis 可以在 AOF 文件体积变得过大时，自动地在后台对 AOF 进行重写**：重写后的新 AOF 文件包含了恢复当前数据集所需的最小命令集合。整个重写操作是绝对安全的，因为 Redis 在创建新 AOF 文件的过程中，会继续将命令追加到现有的 AOF 文件里面，即使重写过程中发生停机，现有的 AOF 文件也不会丢失。而一旦新 AOF 文件创建完毕，Redis 就会从旧 AOF 文件切换到新 AOF 文件，并开始对新 AOF 文件进行追加操作。
4. AOF 日志文件的命令可读性很强，非常适合做灾难性的误删除的紧急恢复

- AOF 文件有序地保存了对数据库执行的所有写入操作，这些写入操作以 Redis 协议的格式保存，因此 AOF 文件的内容非常容易被人读懂，对文件进行分析（parse）也很轻松。导出（export）AOF 文件也非常简单：举个例子，如果你不小心执行了 [FLUSHALL](#) 命令，但只要 AOF 文件未被重写，那么只要停止服务器，移除 AOF 文件末尾的 [FLUSHALL](#) 命令，并重启 Redis，就可以将数据集恢复到 [FLUSHALL](#) 执行之前的状态。

缺点

1. 对于相同的数据集来说，**AOF 文件的体积通常要大于 RDB 文件的体积。**
2. **AOF 开启后，支持的写 QPS 会比 RDB 支持的写 QPS 低**，因为 AOF 一般会配置成每秒 fsync 一次日志文件，当然，每秒一次 fsync，性能也还是很高的。（如果实时写入，那么 QPS 会大降，redis 性能会大大降低）
 - 根据所使用的 `fsync` 策略，AOF 的速度可能会慢于 RDB。在一般情况下，每秒 `fsync` 的性能依然非常高，而关闭 `fsync` 可以让 AOF 的速度和 RDB 一样快，即使在高负荷之下也是如此。不过在处理巨大的写入载入时，RDB 可以提供更有保证的最大延迟时间（latency）。
3. AOF 在过去曾经发生过这样的 bug：因为个别命令的原因，导致 AOF 文件在重新载入时，无法将数据集恢复成保存时的原样。（举个例子，阻塞命令 [BRPOPLPUSH](#) 就曾经引起过这样的 bug。）测试套件里为这种情况添加了测试：它们会自动生成随机的、复杂的数据集，并通过重新载入这些数据来确保一切正常。虽然这种 bug 在 AOF 文件中并不常见，但是对比来说，RDB 几乎是不可能出现这种 bug 的

AOF 文件重写的触发

分为手动触发和自动触发：

- 手动触发：直接调用 `bgrewriteaof` 命令，该命令的执行与 `bgsave` 有些类似：都是 fork 子进程进行具体的工作，且都只有在 fork 时阻塞。
- 自动触发：根据 `auto-aof-rewrite-min-size` 和 `auto-aof-rewrite-percentage` 参数，以及 `aof_current_size` 和 `aof_base_size` 状态确定触发时机。
 - `auto-aof-rewrite-min-size`：执行 AOF 重写时，文件的最小体积，默认值为 64MB。
 - `auto-aof-rewrite-percentage`：执行 AOF 重写时，当前 AOF 大小（即 `aof_current_size`）和上一次重写时 AOF 大小（`aof_base_size`）的比值。
- 关于文件重写的流程，有两点需要特别注意：
 - (1) 重写由父进程 fork 子进程进行；
 - (2) 重写期间 Redis 执行的写命令，需要追加到新的 AOF 文件中，为此 Redis 引入了 `aof_rewrite_buf` 缓存。

RDB 和 AOF的选择

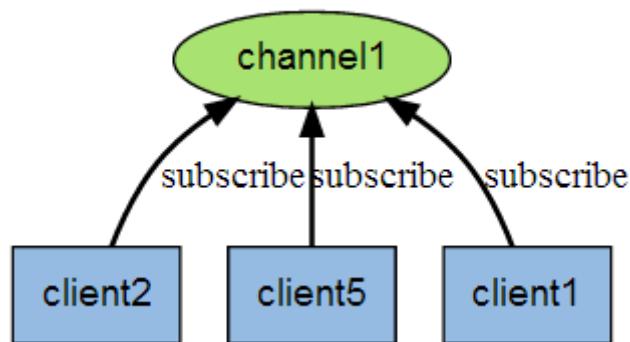
- 如果不需要，也可以不做缓存
- 不要仅仅使用 RDB，那样会导致丢失很多数据
- 也不要仅仅使用 AOF，因为那样有两个问题：第一，你通过 AOF 做冷备，没有 RDB 做冷备来的恢复速度更快；第二，RDB 每次简单粗暴生成数据快照，更加健壮，可以避免 AOF 这种复杂的备份和恢复机制的 bug。
- redis 支持同时开启两种持久化方式，我们可以**综合使用 AOF 和 RDB 两种持久化机制**，用 AOF 来保证数据不丢失，作为数据恢复的第一选择；用 RDB 来做不同程度的冷备，在 AOF 文件都丢失或损坏不可用的时候，还可以使用 RDB 来进行快速的数据恢复。
- 在主从复制中 RDB文件作为备用，而AOF几乎不使用？

Redis发布订阅

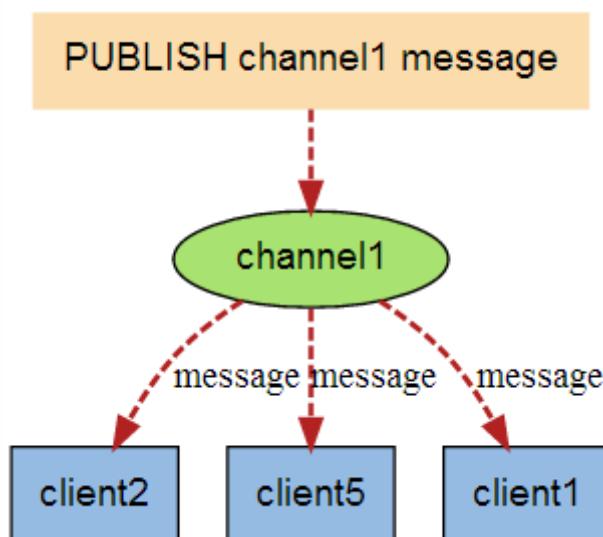
Redis 发布订阅 (pub/sub) 是一种消息通信模式：发送者 (pub) 发送消息，订阅者 (sub) 接收消息。

Redis 客户端可以订阅任意数量的频道。

下图展示了频道 channel1 ， 以及订阅这个频道的三个客户端 —— client2 、 client5 和 client1 之间的关系：



当有新消息通过 PUBLISH 命令发送给频道 channel1 时， 这个消息就会被发送给订阅它的三个客户端：



实例

```
# 第一个 redis-cli 客户端（订阅者）订阅频道 runoobChat
127.0.0.1:6379> subscribe runoobChat
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "runoobChat"
3) (integer) 1
```

```
# 第二个 redis-cli 客户端（发送者）在同一个频道 runoobChat 发布两次消息，订阅者就能接收到消息
127.0.0.1:6379> publish runoobChat "hello,redis!"
(integer) 1
127.0.0.1:6379> publish runoobChat "Learn redis"
(integer) 1
```

```
# 第一个 redis-cli 客户端会显示如下消息
1) "message"
2) "runoobChat"
3) "hello,redis!"
1) "message"
2) "runoobChat"
3) "Learn redis"
```

应用

1. 实时消息系统
2. 实时聊天
3. 订阅、关注系统

复杂的场景应该使用消息中间件 RabbitMQ、kafka

Redis 发布订阅命令

下表列出了 redis 发布订阅常用命令：

序号	命令及描述
1	[PSUBSCRIBE pattern ...] 订阅一个或多个符合给定模式的频道。
2	PUBSUB subcommand [argument [argument ...]] 查看订阅与发布系统状态。
3	PUBLISH channel message 将信息发送到指定的频道。
4	PUNSUBSCRIBE [pattern [pattern ...]] 退订所有给定模式的频道。
5	[SUBSCRIBE channel channel ...] 订阅给定的一个或多个频道的信息。
6	UNSUBSCRIBE [channel [channel ...]] 指退订给定的频道。

Redis主从复制

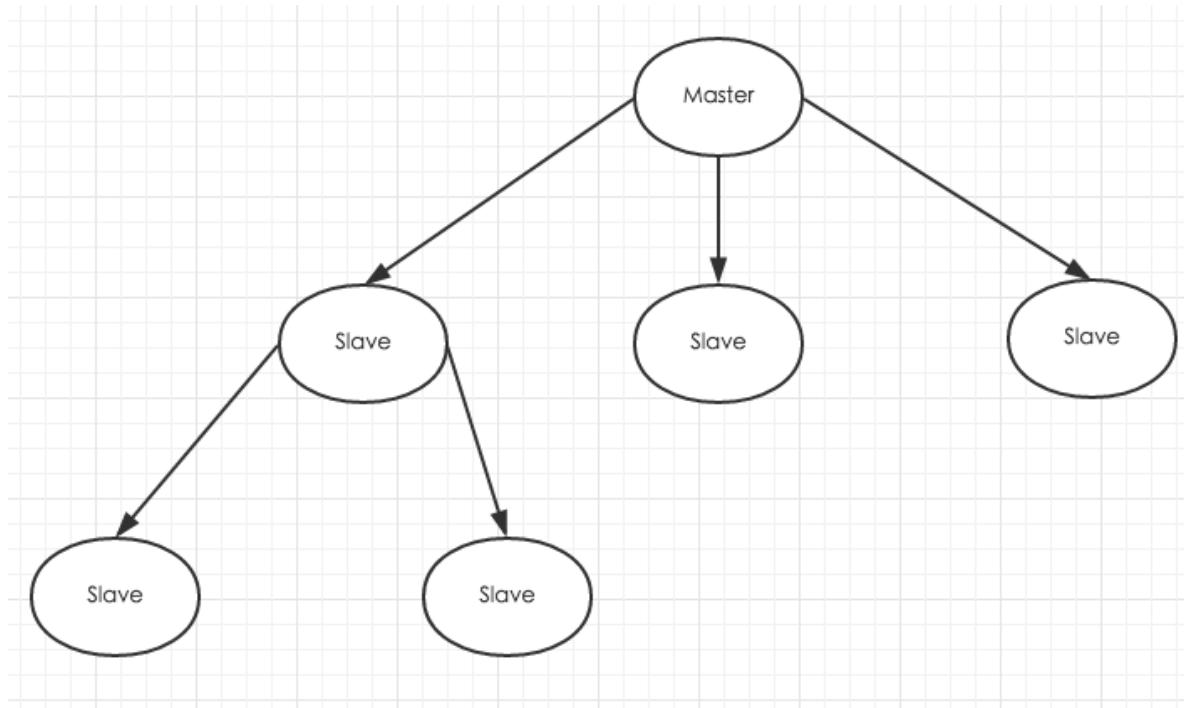
文章：<https://www.cnblogs.com/daofaziran/p/10978628.html>、<https://www.cnblogs.com/kismet/v/p/9236731.html#t3>

1、概述

主从复制，是指将一台Redis服务器的数据，复制到其他的Redis服务器。前者称为主节点(master)，后者称为从节点(slave)；数据的复制是单向的，只能由主节点到从节点。Master以写为主，Slave以读为主。

默认情况下，每台Redis服务器都是主节点；且一个主节点可以有多个从节点(或没有从节点)，但一个从节点只能有一个主节点。

Redis的主从结构可以采用一主多从或者级联结构，Redis主从复制可以根据是否是全量分为全量同步和增量同步。下图为级联结构。



主从复制的作用

主从复制的作用主要包括：

1. 数据冗余：**主从复制实现了数据的热备份**，是持久化之外的一种数据冗余方式。
2. 故障恢复：当主节点出现问题时，可以由从节点提供服务，**实现快速的故障恢复**；实际上是一种服务的冗余。
3. 负载均衡：在主从复制的基础上，**配合读写分离，可以由主节点提供写服务，由从节点提供读服务**（即写Redis数据时应用连接主节点，读Redis数据时应用连接从节点），分担服务器负载；尤其是在写少读多的场景下，通过多个从节点分担读负载，可以大大提高Redis服务器的并发量。
4. 高可用基石：除了上述作用以外，**主从复制还是哨兵和集群能够实施的基础**，因此说主从复制是Redis高可用的基础。

为什么要使用主从复制？

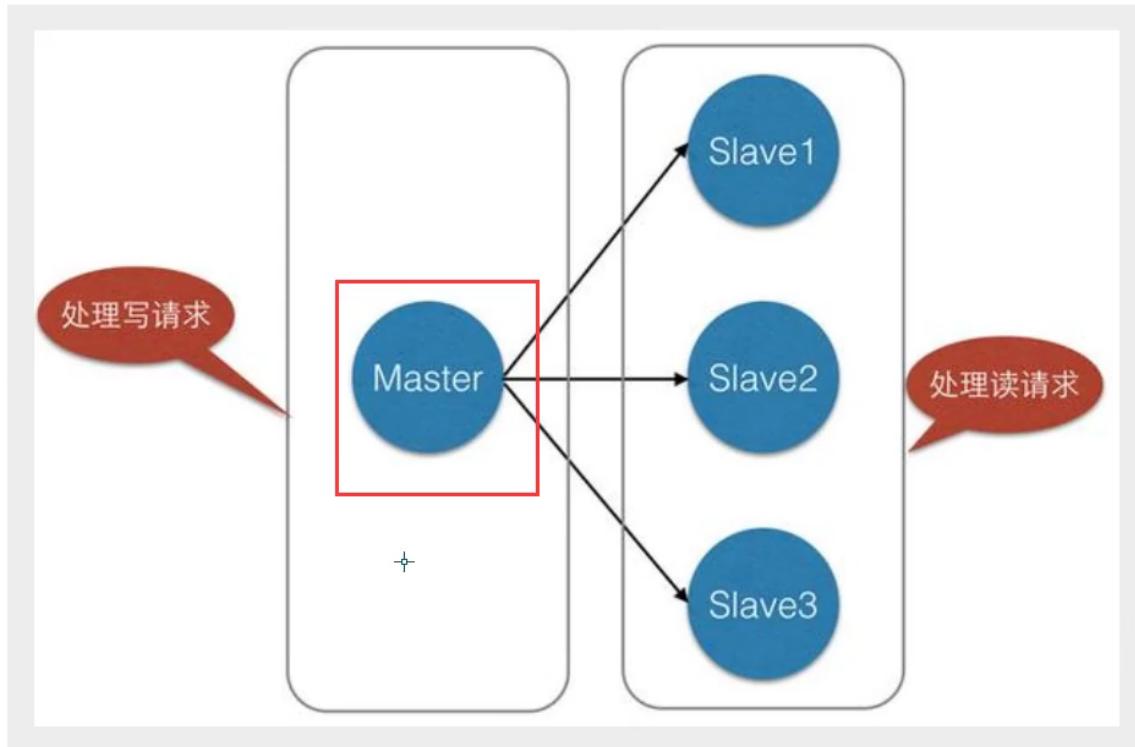
主从复制可以实现读写分离，减缓服务器压力

一般来说，要将Redis运用于工程项目中，只使用一台Redis是万万不能的，原因如下：

- 1、从结构上，单个Redis服务器会发生单点故障，并且一台服务器需要处理所有的请求负载，压力较大；
- 2、从容量上，单个Redis服务器内存容量有限，就算一台Redis服务器内存容量为256G，也不能将所有内存用作Redis存储内存；

一般来说，单台Redis最大使用内存不应该超过20G。

电商网站上的商品，一般都是一次上传，无数次浏览的，说专业点也就是“多读少写”。



2、如何使用主从复制

2.1、配置redis服务

docker开启redis服务

```
docker run -p 6380:6379 -p 26380:26379 --name redis6380 -d redis # 从节点  
docker run -p 6381:6379 -p 26381:26379 --name redis6381 -d redis # 从节点
```

```
docker exec -it redis6380 /bin/bash  
redis-cli  
docker exec -it redis6381 /bin/bash  
redis-cli
```

linux主机配置redis服务 (选读)

Docker安装的不需要改配置文件，将端口映射和容器名称改一下就可以了

1. copy

```
cp redis.conf redis79.conf  
cp redis.conf redis80.conf  
cp redis.conf redis81.conf
```

2. 修改配置文件

1. 修改 端口号
2. 修改pid文件名
3. 修改日志文件名
4. 修改 rbd文件名

3. 启动redis服务

```
redis-server redis79.conf  
redis-server redis80.conf  
redis-server redis81.conf
```

4. 查看进程

```
ps -ef|grep redis
```

2.2、配置从节点

只需要配置从节点

命令行配置 (重启后失效)

```
#####  
127.0.0.1:6379> info replication # 查看当前库信息 (redis节点信息)  
# Replication  
role:master # 角色  
connected_slaves:0 # 从节点个数  
master_failover_state:no-failover  
master_replid:4a33e8de810874366ce0787a30c64b6d64308986  
master_replid2:0000000000000000000000000000000000000000000000000000000000000000  
master_repl_offset:0  
second_repl_offset:-1  
replog_backlog_active:0  
replog_backlog_size:1048576  
replog_backlog_first_byte_offset:0  
replog_backlog_histlen:0  
#####  
  
127.0.0.1:6379> slaveof 172.17.0.3 6379 # 认主机  
OK  
  
#####主机信息#####  
127.0.0.1:6379> info replication
```

```
# Replication
role:master
connected_slaves:2
slave0:ip=172.17.0.8,port=6379,state=online,offset=98,lag=1
slave1:ip=172.17.0.9,port=6379,state=online,offset=98,lag=1
master_failover_state:no-failover
master_replid:017c5c806d6a1d3ec684270fff1b0520aa91b44e
master_replid2:0000000000000000000000000000000000000000000000000000000000000000
master_repl_offset:98
second_repl_offset:-1
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:1
repl_backlog_histlen:98

#####
#从机信息#####
127.0.0.1:6379> info replication
# Replication
role:slave
master_host:172.17.03
master_port:6379
master_link_status:up
master_last_io_seconds_ago:2
master_sync_in_progress:0
slave_read_repl_offset:168
slave_repl_offset:168
slave_priority:100
slave_read_only:1
replica_announced:1
connected_slaves:0
master_failover_state:no-failover
master_replid:017c5c806d6a1d3ec684270fff1b0520aa91b44e
master_replid2:0000000000000000000000000000000000000000000000000000000000000000
master_repl_offset:168
second_repl_offset:-1
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:1
repl_backlog_histlen:168
```

修改配置文件

```
replicaof <masterip> <masterport>
```

```

'1 ##### REPLICATION #####
'2
'3 # Master-Replica replication. Use replicaof to make a Redis instance a copy of
'4 # another Redis server. A few things to understand ASAP about Redis replication.
'5 #
'6 # +-----+ +-----+
'7 # | Master | --> | Replica |
'8 # | (receive writes) | | (exact copy) |
'9 # +-----+ +-----+
'0 #
'1 # 1) Redis replication is asynchronous, but you can configure a master to
'2 # stop accepting writes if it appears to be not connected with at least
'3 # a given number of replicas.
'4 # 2) Redis replicas are able to perform a partial resynchronization with the
'5 # master if the replication link is lost for a relatively small amount of
'6 # time. You may want to configure the replication backlog size (see the next
'7 # sections of this file) with a sensible value depending on your needs.
'8 # 3) Replication is automatic and does not need user intervention. After a
'9 # network partition replicas automatically try to reconnect to masters
'10 # and resynchronize with them.
'11 #
'12 # replicaof <masterip> <masterport>
'13
'14 # If the master is password protected (using the "requirepass" configuration

```

2.3、宕机后手动配置主机

如果主节点挂了，那么只能**手动**使用命令启用一个新的主节点。

命令：

```
SLAVEOF no one
```

可以设置一个节点为主机，其他的节点就可以**手动连接**到最新的这个主节点

3、主从复制实现原理

Slave启动成功连接到master后会发送一个sync命令

Master接到命令，启动后台的存盘进程，同时收集所有接收到的用于修改数据集命令，在后台进程执行完毕之后，master将传送整个数据文件到slave，并完成一次完全同步（**全量复制**）。

全量复制: slave服务在接收到数据库文件数据后，将其存盘并加载到内存中。

增量复制: Master继续将新的所有收集到的修改命令依次传给slave，完成同步
但是只要是重新连接master，一次完全同步（全量复制）将被自动执行。

主从复制过程大体可以分为3个阶段：连接建立阶段（即准备阶段）、数据同步阶段、命令传播阶段；

3.1、连接建立阶段

该阶段的主要作用是在主从节点之间建立连接，为数据同步做好准备。

步骤1：保存主节点信息

从节点服务器内部维护了两个字段，即masterhost和masterport字段，用于存储主节点的ip和port信息。

需要注意的是，**slaveof是异步命令**，从节点完成主节点ip和port的保存后，向发送slaveof命令的客户端直接返回OK，实际的复制操作在这之后才开始进行。

步骤2：建立socket连接

从节点每秒1次调用复制定时函数replicationCron()，如果发现了有主节点可以连接，便会根据主节点的ip和port，创建socket连接。如果连接成功，则：

从节点：为该socket建立一个专门处理复制工作的文件事件处理器，负责后续的复制工作，如接收RDB文件、接收命令传播等。

主节点：接收到从节点的socket连接后（即accept之后），为该socket创建相应的客户端状态，并将从节点看做是连接到主节点的一个客户端，后面的步骤会以从节点向主节点发送命令请求的形式来进行。

步骤3：发送ping命令

从节点成为主节点的客户端之后，发送ping命令进行首次请求，目的是：检查socket连接是否可用，以及主节点当前是否能够处理请求。

从节点发送ping命令后，可能出现3种情况：

- (1) 返回pong：说明socket连接正常，且主节点当前可以处理请求，复制过程继续。
- (2) 超时：一定时间后从节点仍未收到主节点的回复，说明socket连接不可用，则从节点断开socket连接，并重连。
- (3) 返回pong以外的结果：如果主节点返回其他结果，如正在处理超时运行的脚本，说明主节点当前无法处理命令，则从节点断开socket连接，并重连。

步骤4：身份验证

如果从节点中设置了masterauth选项，则从节点需要向主节点进行身份验证；没有设置该选项，则不需要验证。从节点进行身份验证是通过向主节点发送auth命令进行的，auth命令的参数即为配置文件中的masterauth的值。

如果主节点设置密码的状态，与从节点masterauth的状态一致（一致是指都存在，且密码相同，或者都不存在），则身份验证通过，复制过程继续；如果不一致，则从节点断开socket连接，并重连。

步骤5：发送从节点端口信息

身份验证之后，从节点会向主节点发送其监听的端口号（前述例子中为6380），主节点将该信息保存到该从节点对应的客户端的slave_listening_port字段中；该端口信息除了在主节点中执行info Replication时显示以外，没有其他作用。

3.2、数据同步阶段

主从节点之间的连接建立以后，便可以开始进行数据同步，该阶段可以理解为从节点数据的初始化。具体执行的方式是：从节点向主节点发送psync命令（Redis2.8以前是sync命令），开始同步。

数据同步阶段是主从复制最核心的阶段，根据主从节点当前状态的不同，**可以分为全量复制和部分复制**，下面会有一章专门讲解这两种复制方式以及psync命令的执行过程，这里不再详述。

需要注意的是，在**数据同步阶段之前**，从节点是主节点的客户端，主节点不是从节点的客户端；而到了**这一阶段及以后**，主从节点互为客户端。原因在于：在此之前，主节点只需要响应从节点的请求即可，不需要主动发请求，而在数据同步阶段和后面的命令传播阶段，主节点需要主动向从节点发送请求（如推送缓冲区中的写命令），才能完成复制。

3.3、命令传播阶段

数据同步阶段完成后，主从节点进入命令传播阶段；在这个阶段主节点将自己执行的写命令发送给从节点，从节点接收命令并执行，从而保证主从节点数据的一致性。

在命令传播阶段，除了发送写命令，主从节点还维持着心跳机制：PING和REPLCONF ACK。由于心跳机制的原理涉及部分复制，因此将在介绍了部分复制的相关内容后单独介绍该心跳机制。

延迟与不一致

需要注意的是，命令传播是异步的过程，即主节点发送写命令后并不会等待从节点的回复；因此实际上主从节点之间很难保持实时的一致性，延迟在所难免。数据不一致的程度，与主从节点之间的网络状况、主节点写命令的执行频率、以及主节点中的repl-disable-tcp-nodelay配置等有关。

repl-disable-tcp-nodelay no：该配置作用于命令传播阶段，控制主节点是否禁止与从节点的TCP_NODELAY；默认no，即不禁止TCP_NODELAY。当设置为yes时，TCP会对包进行合并从而减少带宽，但是发送的频率会降低，从节点数据延迟增加，一致性变差；具体发送频率与Linux内核的配置有关，默认配置为40ms。当设置为no时，TCP会立马将主节点的数据发送给从节点，带宽增加但延迟变小。

一般来说，只有当应用对Redis数据不一致的容忍度较高，且主从节点之间网络状况不好时，才会设置为yes；多数情况使用默认值no。

4、全量复制与增量复制

全量复制

Redis全量复制一般发生在Slave初始化阶段，这时Slave需要将Master上的所有数据都复制一份。具体步骤如下：

- 从服务器连接主服务器，发送SYNC命令；
- 主服务器接收到SYNC命令后，开始执行BGSAVE命令生成RDB文件并使用缓冲区记录此后执行的所有写命令；
- 主服务器BGSAVE执行完后，向所有从服务器发送快照文件，并在发送期间继续记录被执行的写命令；
- 从服务器收到快照文件后丢弃所有旧数据，载入收到的快照；
- 主服务器快照发送完毕后开始向从服务器发送缓冲区中的写命令；
- 从服务器完成对快照的载入，开始接收命令请求，并执行来自主服务器缓冲区的写命令；



完成上面几个步骤后就完成了从服务器数据初始化的所有操作，从服务器此时可以接收来自用户的读请求。

增量复制

Redis增量复制是指Slave初始化后开始正常工作时主服务器发生的写操作同步到从服务器的过程。增量复制的过程主要是主服务器每执行一个写命令就会向从服务器发送相同的写命令，从服务器接收并执行收到的写命令。

5、Redis主从同步策略

主从刚刚连接的时候，进行全量同步；全量同步结束后，进行增量同步。当然，如果有需要，slave 在任何时候都可以发起全量同步。redis 策略是，无论如何，首先会尝试进行增量同步，如不成功，要求从机进行全量同步。

注意点

如果多个Slave断线了，需要重启的时候，因为只要Slave启动，就会发送sync请求和主机全量同步，当多个同时出现的时候，可能会导致Master IO剧增宕机。

Redis主从复制的配置十分简单，它可以使从服务器是主服务器的完全拷贝。需要清楚Redis主从复制的几点重要内容：

- 1) Redis使用异步复制。但从Redis 2.8开始，从服务器会周期性的应答从复制流中处理的数据量。
- 2) 一个主服务器可以有多个从服务器。
- 3) 从服务器也可以接受其他从服务器的连接。除了多个从服务器连接到一个主服务器之外，多个从服务器也可以连接到一个从服务器上，形成一个图状结构。
- 4) Redis主从复制不阻塞主服务器端。也就是说当若干个从服务器在进行初始同步时，主服务器仍然可以处理请求。
- 5) 主从复制也不阻塞从服务器端。当从服务器进行初始同步时，它使用旧版本的数据来应对查询请求，假设你在redis.conf配置文件是这么配置的。
否则的话，你可以配置当复制流关闭时让从服务器给客户端返回一个错误。但是，当初始同步完成后，需要删除旧的数据集和加载新的数据集，在这个短暂的时间内，从服务器会阻塞连接进来的请求。
- 6) 主从复制可以用来增强扩展性，使用多个从服务器来处理只读的请求（比如，繁重的排序操作可以放到从服务器去做），也可以简单的用做数据冗余。
- 7) 使用主从复制可以为主服务器免除把数据写入磁盘的消耗：在主服务器的redis.conf文件中配置“避免保存”（注释掉所有“保存”命令），然后连接一个配置为“进行保存”的从服务器即可。但是这个配置要确保主服务器不会自动重启

6、Redis 主从复制特点

- 1) 采用异步复制；
- 2) 一个主redis可以含有多个从redis；
- 3) 每个从redis可以接收来自其他从redis服务器的连接；
- 4) 主从复制对于主redis服务器来说是非阻塞的，这意味着当从服务器在进行主从复制同步过程中，主redis仍然可以处理外界的访问请求；
- 5) 主从复制对于从redis服务器来说也是非阻塞的，这意味着，即使从redis在进行主从复制过程中也可以接受外界的查询请求，只不过这时候从redis返回的是以前老的数据，

如果你不想这样，那么在启动redis时，可以在配置文件中进行设置，那么从redis在复制同步过程中来自外界的查询请求都会返回错误给客户端；（虽然说主从复制过程中

对于从redis是非阻塞的，但是当从redis从主redis同步过来最新的数据后还需要将新数据加载到内存

中，在加载到内存的过程中是阻塞的，在这段时间内的请求将会被阻，

但是即使对于大数据集，加载到内存的时间也是比较多的)；

6) 主从复制提高了redis服务的扩展性，避免单个redis服务器的读写访问压力过大的问题，同时也可以给为数据备份及冗余提供一种解决方案；

7) 为了缓解主redis服务器写磁盘压力带来的开销，可以配置让主redis不在将数据持久化到磁盘，而是通过连接让一个配置的从redis服务器及时的将相关数据持久化到磁盘，

不过这样会存在一个问题，就是主redis服务器一旦重启，因为主redis服务器数据为空，这时候通过主从同步可能导致从redis服务器上的数据也被清空；

7、主从同步中需要注意几个问题

1) 在上面的全量同步过程中，master会将数据保存在rdb文件中然后发送给slave服务器，但是如果master上的磁盘空间有限怎么办呢？那么此时全部同步对于master来说

将是一份十分有压力的操作了。此时可以通过无盘复制来达到目的，由master直接开启一个socket将rdb文件发送给slave服务器。（无盘复制一般应用在磁盘空间有限但是网络状态良好的情况下）

2) 主从复制结构，一般slave服务器不能进行写操作，但是这不是死的，之所以这样是为了更容易的保证主和各个从之间数据的一致性，如果slave服务器上数据进行了修改，那么要保证所有主从服务器都能一致，可能在结构上和处理逻辑上更为负责。不过你也可以通过配置文件让从服务器支持写操作。（不过所带来的影响还得自己承担哦。。。）

3) 主从服务器之间会定期进行通话，但是如果master上设置了密码，那么如果不给slave设置密码就会导致slave不能跟master进行任何操作，所以如果你的master服务器上有密码，那么也给slave相应的设置一下密码吧（通过设置配置文件中的masterauth）；

4) 关于slave服务器上过期键的处理，由master服务器负责键的过期删除处理，然后将相关删除命令已数据同步的方式同步给slave服务器，slave服务器根据删除命令删除本地的key。

8、链路式主从



中间节点依然还是从节点，即无法写入。

如果主节点挂了，那么只能**手动**使用命令启用一个新的主节点。

命令：

```
SLAVEOF no one
```

可以设置一个节点为主机，其他的节点就可以***手动连接***到最新的这个主节点

9、主从复制优缺点

优点：

1. 读写分离，提高效率
2. 主节点负责写操作，从节点负责读操作；如果写少读多场景，配置多个从节点的话，效率非常高
3. 数据热备份，提供多个副本。
4. 从节点宕机，影响较小

缺点：

1. 主节点故障，集群则无法进行工作，可用性比较低，从节点升主节点需要人工手动干预。
2. 因为只有主节点能进行写操作，一旦主节点宕机，整个服务就无法使用。当然此时从节点仍可以进行读操作，但是对于整个服务流程来说，是无法使用的。
3. Master的写的压力难以降低。
4. 如果写操作比较多，那么只有一个主节点的话，无法分担压力。
5. 主节点存储能力受到单机限制。
6. 主节点只能有一个，因此单节点内存大小不会太大，因此存储数据量受限。
7. 主从数据同步，可能产生部分的性能影响甚至同步风暴。
 - 风暴问题，对于任何集群分布式来说都存在，要合理分布节点。

Redis 哨兵模式

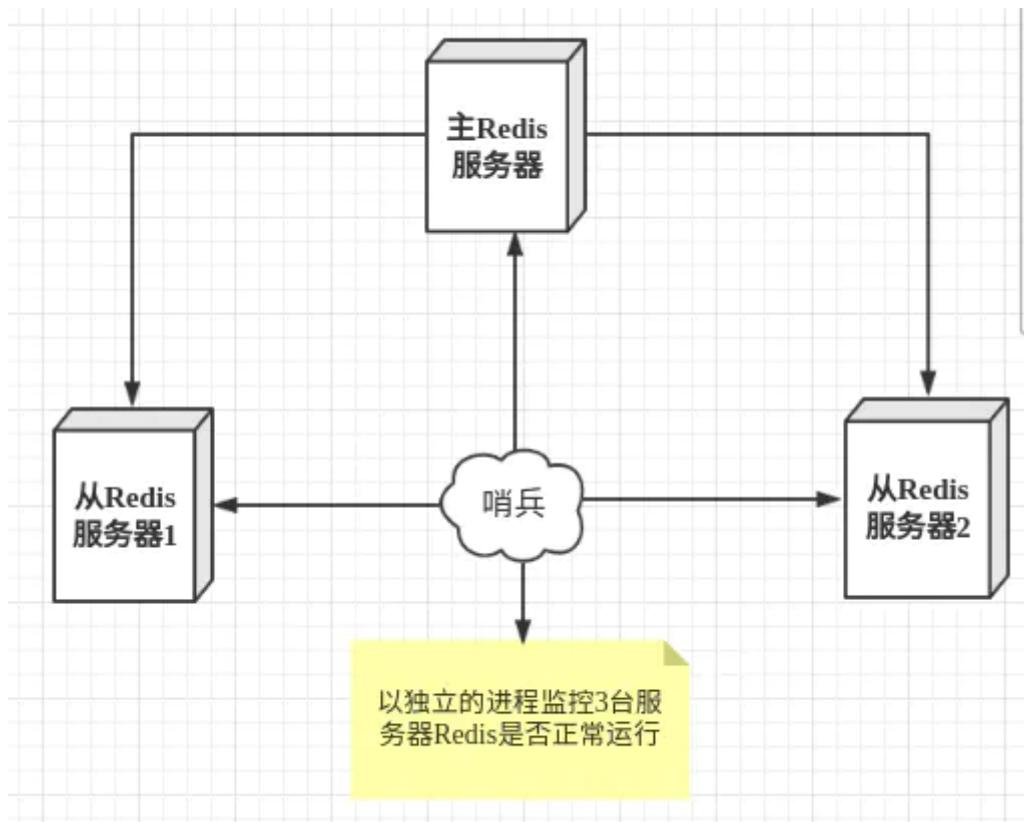
在redis2.4之前的故障转移，是需要人为干预才能实现的，redis2.4之后引入了**Sentinel**。主服务器一段长时间内无法正常工作时，sentinel将自主发现、自动执行故障转移。sentinel的引入，最大程度的保证了redis的**高可用**。

总结：

Redis主从复制的缺点：没有办法对master进行动态选举，需要使用Sentinel机制完成动态选举

1、哨兵模式概述

哨兵模式是一种特殊的模式，首先Redis提供了哨兵的命令，哨兵是一个独立的进程，作为进程，它会独立运行。其原理是哨兵通过发送命令，等待Redis服务器响应，从而监控运行的多个Redis实例。

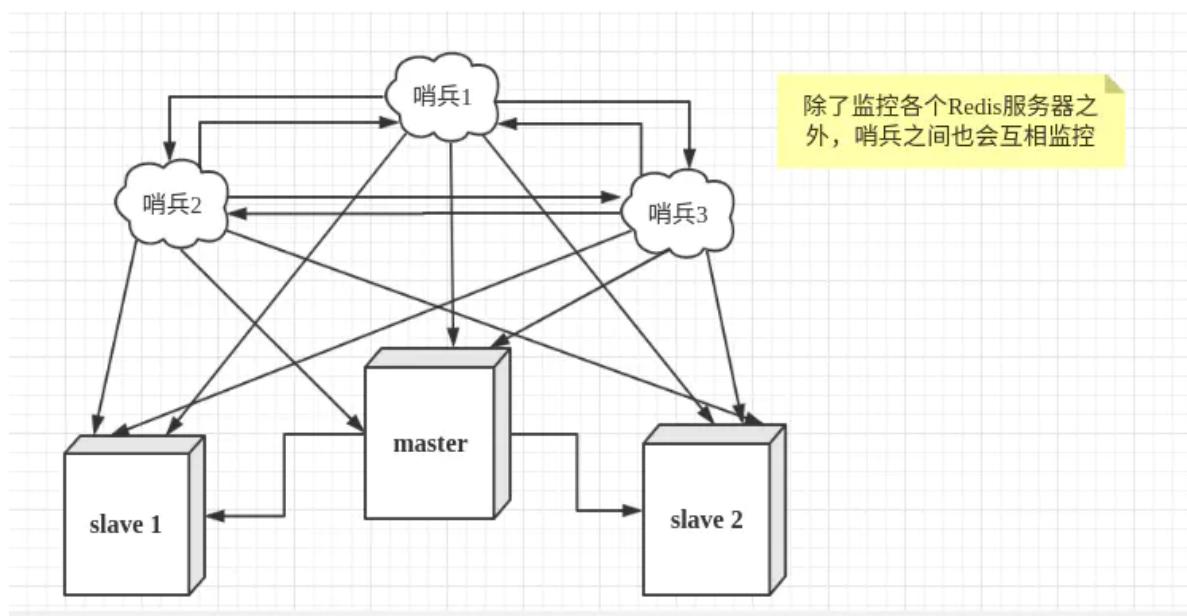


这里的哨兵有两个作用

- 通过发送命令，让Redis服务器返回监控其运行状态，包括主服务器和从服务器。
- 当哨兵监测到master宕机，会自动将slave切换成master，然后通过**发布订阅模式**通知其他的从服务器，修改配置文件，让它们切换主机。

然而一个哨兵进程对Redis服务器进行监控，可能会出现问题，为此，我们可以使用多个哨兵进行监控。各个哨兵之间还会进行监控，这样就形成了多哨兵模式。

Sentinel会自动监控master、 slave、 其他sentinel的状态，当发现master出现故障， sentinel会将此master标记下线，并选择状态较好的slave为新任master，其他slave的复制源切换到新任master服务上，原master设置成新任master的slave。



故障切换 (failover) 的过程:

假设主服务器宕机，哨兵1先检测到这个结果，系统并不会马上进行failover过程，仅仅是哨兵1主观的认为主服务器不可用，这个现象成为主观下线。当后面的哨兵也检测到主服务器不可用，并且数量达到一定值时，那么哨兵之间就会进行一次投票，投票的结果由一个哨兵发起，进行failover操作。切换成功后，就会通过发布订阅模式，让各个哨兵把自己监控的从服务器实现切换主机，这个过程称为客观下线。这样对于客户端而言，一切都是透明的。

2、哨兵模式作用

1. **监控(Monitoring)**: 哨兵(sentinel) 会不断地检查你的Master和Slave是否运作正常。
2. **提醒(Notification)**: 当被监控的某个Redis节点出现问题时，哨兵(sentinel) 可以通过 API 向管理员或者其他应用程序发送通知。
3. **自动故障迁移(Automatic failover)**: 当一个Master不能正常工作时，哨兵(sentinel) 会开始一次自动故障迁移操作。

- 它会将失效Master的其中一个Slave升级为新的Master，并让失效Master的其他Slave改为复制新的Master；

当客户端试图连接失效的Master时，集群也会向客户端返回新Master的地址，使得集群可以使用现在的Master替换失效Master。

Master和Slave服务器切换后，Master的redis.conf、Slave的redis.conf和sentinel.conf的配置文件的内容都会发生相应的改变，即，Master主服务器的redis.conf配置文件中会多一行 slaveof的配置，sentinel.conf的监控目标会随之调换。

3、哨兵模式的工作方式

1. 每个Sentinel (哨兵) 进程以**每秒钟一次**的频率向整个集群中的**Master主服务器，Slave从服务器以及其他Sentinel (哨兵) 进程**发送一个 **PING 命令**。
2. 如果一个实例 (instance) 距离最后一次有效回复 PING 命令的时间超过 down-after-milliseconds 选项所指定的值，则这个实例会被 Sentinel (哨兵) 进程标记为**主观下线 (SDOWN)**。
3. 如果一个Master主服务器被标记为主观下线 (SDOWN)，则正在监视这个Master主服务器的所有 Sentinel (哨兵) 进程要以每秒一次的频率**确认Master主服务器的确进入了主观下线状态**。
4. 当有足够的 Sentinel (哨兵) 进程（大于等于配置文件指定的值）在指定的时间范围内**确认 Master主服务器进入了主观下线状态 (SDOWN)**，则Master主服务器会被标记为**客观下线 (ODOWN)**。
5. 在一般情况下，每个Sentinel (哨兵) 进程会以每 10 秒一次的频率向集群中的所有Master主服务器、Slave从服务器发送 INFO 命令。
6. 当Master主服务器被 Sentinel (哨兵) 进程标记为**客观下线 (ODOWN)** 时，Sentinel (哨兵) 进程向下线的 Master主服务器的所有 Slave从服务器发送 INFO 命令的频率会从 10 秒一次改为每秒一次。
7. 若没有足够数量的 Sentinel (哨兵) 进程同意 Master主服务器下线，Master主服务器的客观下线状态就会被移除。若 Master主服务器重新向 Sentinel (哨兵) 进程发送 PING 命令返回有效回复，Master主服务器的主观下线状态就会被移除。

4、监控服务状态

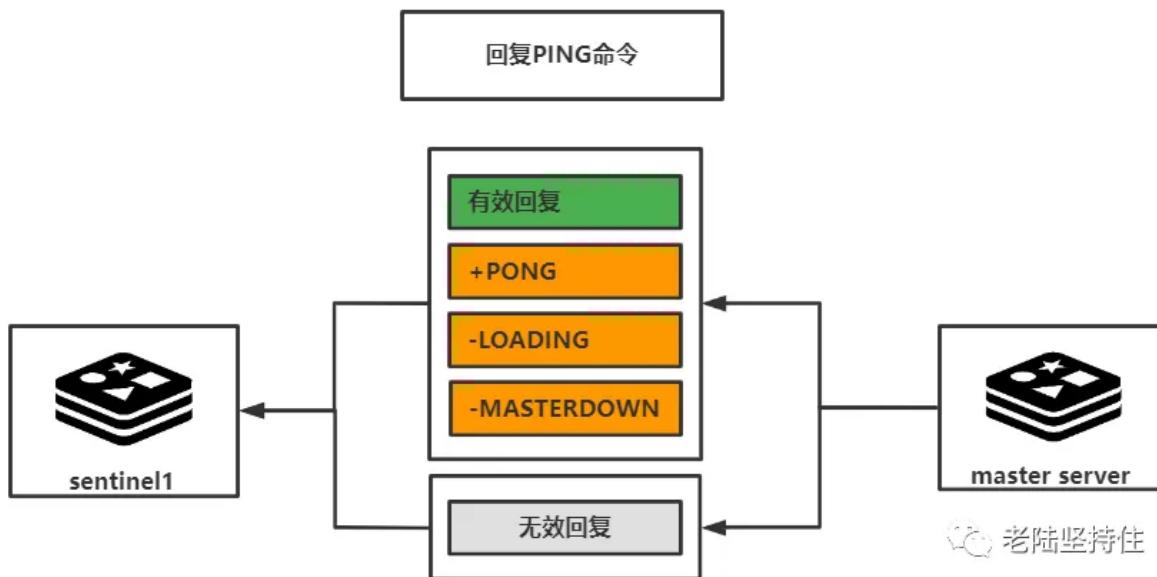
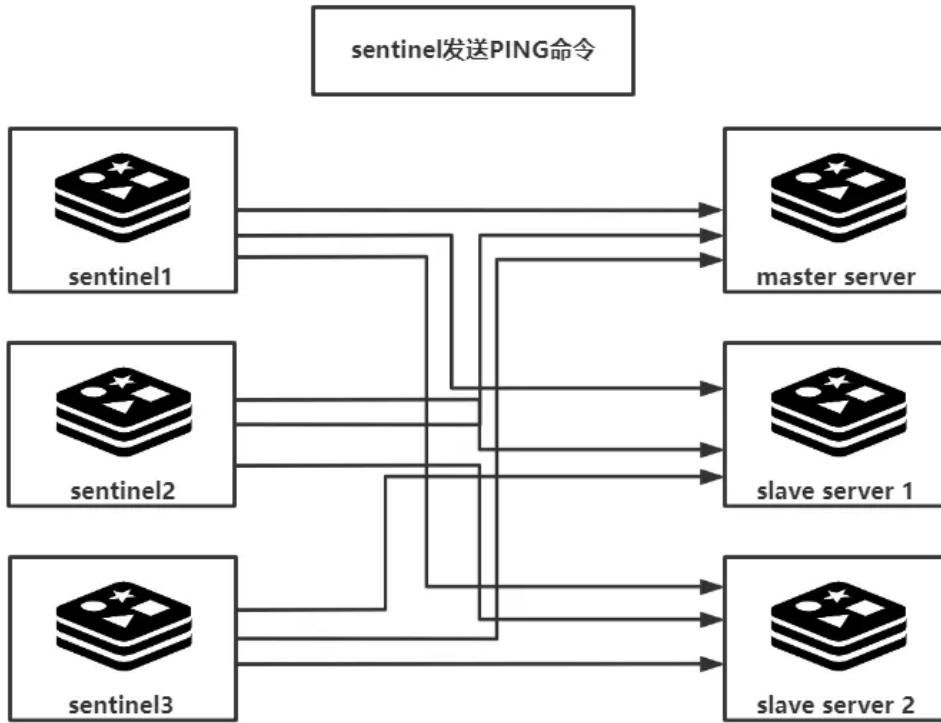
sentinel是如何监控服务状态的呢？又是如何标记下线状态的呢？我们从两种状态展开分析。

主观下线

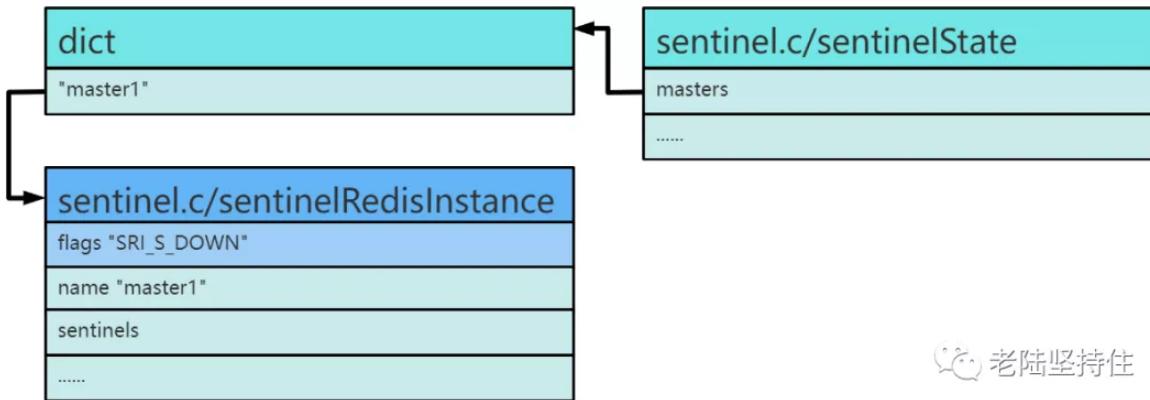
Sentinel会向master、slave以及发现的其他sentinel发送 **PING** 命令， 默认一秒发送一次，对方接收后返回两种回复：

- **有效回复**: 包括运行正常 (+PONG)、正在加载 (-LOADING)、和主机下线 (-MASTERDOWN)；

- 无效回复：不在有效回复内的都是无效回复；



在固定时间内，即 `down-after-milliseconds` 配置的时间内收到的都是无效回复，sentinel就会标记 master为主观下线。与此同时，sentinel会将master数据结构中，对应的 `flags` 属性更新为 `SRI_S_DOWN`，表示被监控的master在当前sentinel中为已下线状态。如下图

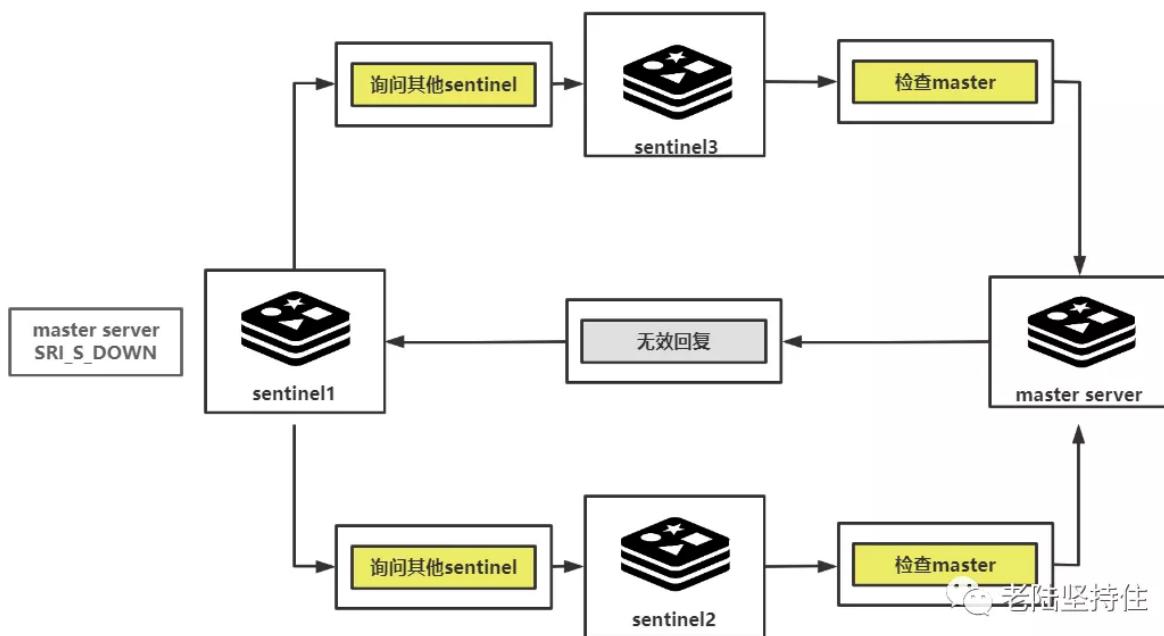


老陆坚持住

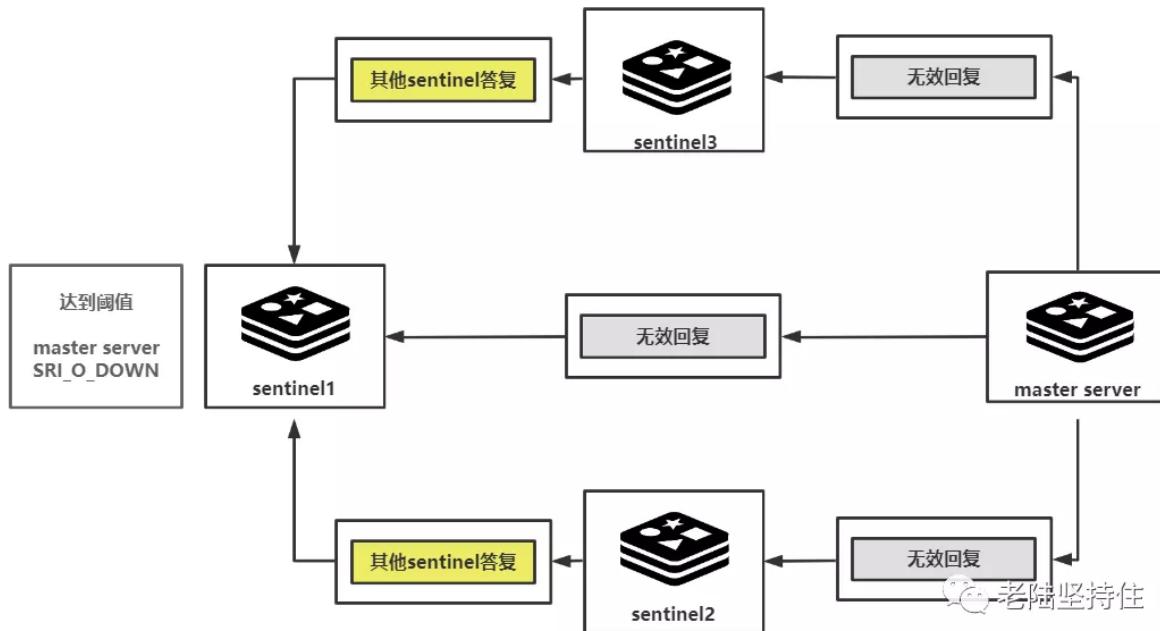
需要注意的是 `down-after-milliseconds` 配置是作用于当前sentinel所监控的所有服务上的，也就是对应master下的slave，以及其他sentinel。另外**每个sentinel可以配置不同** `down-after-milliseconds`，所以判定主观下线的时间也就是不同的。

客观下线

判定master为主观下线的首个sentinel，通过命令询问其他sentinel，目标sentinel收到命令后，会根据参数检查master是否下线，如下图



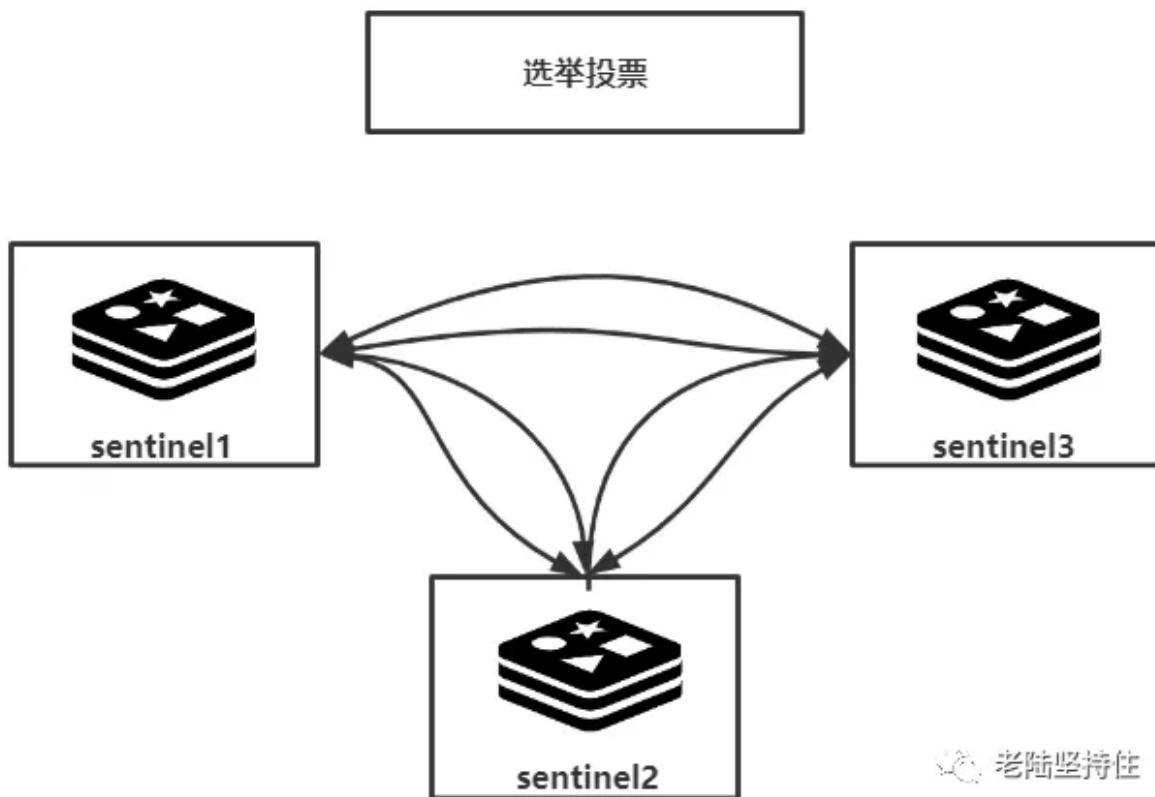
如果收到的也是**无效回复**，会将结果发送给源sentinel，源sentinel会统计包括自己在内的所有**认为master已进入主观下线**的sentinel数量，统计数量如果达到了当前sentinel配置的阈值，会将sentinel的master对应的 `flags` 属性值更改为 `SRI_O_DOWN`，master进入客观下线状态，如下图



注意不同的sentinel可以配置不同的客观下线阈值。

5、选举

当被监控的master被判定为客观下线，接下来就是执行故障转移，那么由哪个sentinel来发送执行故障转移命令呢？Sentinel内部是通过投票、选举来决定出领头sentinel的，所以每个sentinel都会向其他sentinel发送拉票命令



选举是有条件和规则设定的，条件比较多，以下是我汇总后的列表：

- 每个sentinel都有被选择领头sentinel的资格；

- 同一个配置纪元内（配置计数器，即选举后自增），每个sentinel只能选举一次，并且选举成功后，不会再改变；
- 最先判定master为客观下线的sentinel都会要求其他Sentinel将自己设置为局部领头Sentinel；
- 当被半数以上的sentinel支持后，局部领头sentinel就变成了领头sentinel，同一个配置纪元内可能会出现多个局部领头sentinel，但是领头sentinel只会产生一个；

上述规则可以决定领头sentinel，可以看出，投票、选举的过程其实是先到先得的。

我们已经为故障转移做好了准备，那么领头sentinel是如何执行故障转移的呢？

6. 故障转移

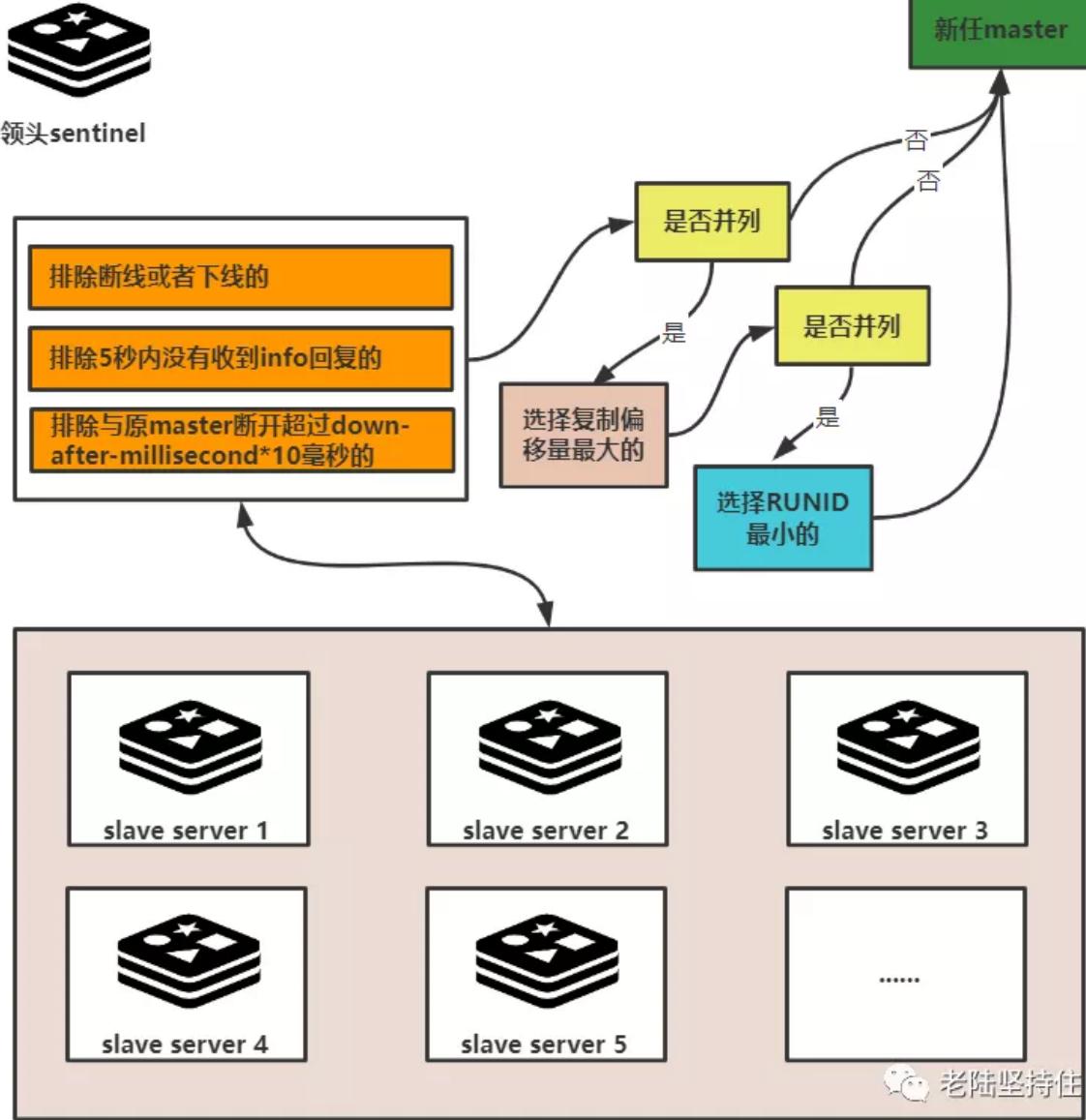
故障转移可以分为三部分，首先就是将master下线并选择从服务器为新任master，然后将新任master重置为其他slave的复制源，最后将被下线的master切换为新任master的slave。

选择

新任master是从原master下的slave中选择的，所以需要选择状态良好、数据完整的slave。领头sentinel的数据结构中保存了原master对应的slave，sentinel会排除状态较差的slave。排除条件如下

- 排除断线或者下线的；
- 排除5秒内没有收到info回复的；
- 排除与原master断开超过 `down-after-millisecond *10毫秒` 的slave。在 `down-after-millisecond` 设置的时长内没有收到有效回复，可以判定master已下线，增加此过滤规则是用于排除slave与原master过早断开连接，保证备选slave的数据是最新的（不理解的同学可以一起讨论）。

经过如上条件的筛选，如果出现并列slave，会对并列slave的复制偏移量进行排序选择最高的slave，如果依然并列的话，就将 `RUNID` 进行排序，`RUNID` 最小的即是新任master，如下图



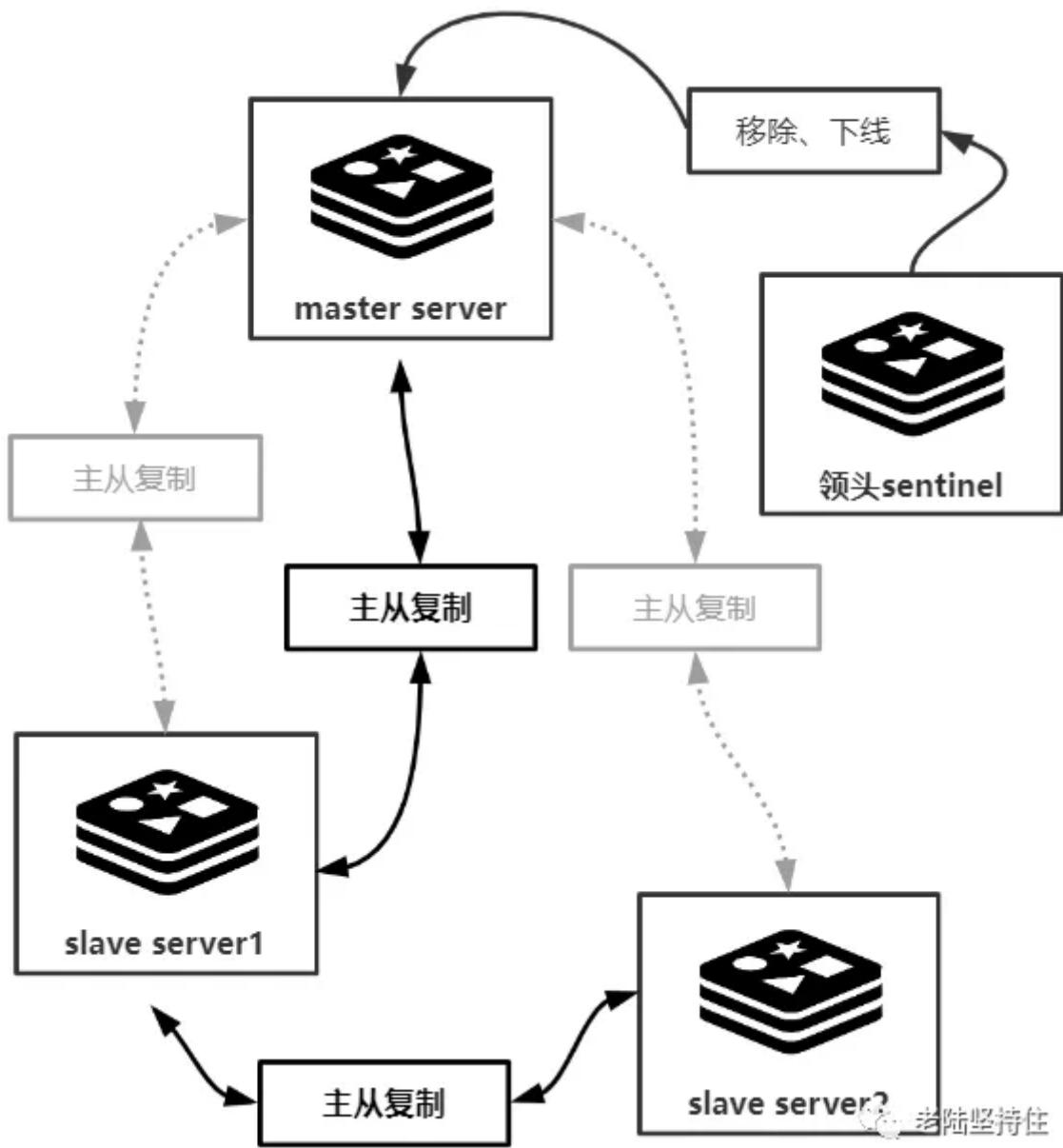
重置

新任master产生后，原master对应的slave会将复制源设置为新任master。

切换

对于已下线、发生故障的master，领头sentinel会将此master设置为slave，复制源设置为新任master。修复上线后就变成了新任master的slave。

故障迁移工作执行完毕，下图展示了迁移的工作流程



7. 测试实例

1. 配置哨兵配置文件 sentinel.conf

```
# sentinel monitor 被监控的名称 host port 1
sentinel monitor redis 127.0.0.1 6379 1
```

“1”这个参数指的是，即多少个主节点检测到主节点有问题就进行故障转移。其实就是有多少个哨兵 (sentinel) 认为这个主节点主观下线才算真正的下线，然后进行故障转移，让其客观下线

主（宕机）--->哨兵A、B、C检测主不可用--->哨兵发起投票、哨兵们投票--->故障转移--->发布订阅--->通知到redis服务器--->切换主机

2. 启动哨兵 `redis-sentinel sentinel.conf`
3. 尝试主动关闭 master `shutdown`，此时查看节点信息 `info replication` 会发现master节点没有改变
4. 等过一段时间后哨兵模式会重新选举新的master，此时查看节点信息 `info replication` 会发现 master节点已经改变

(可选) docker 配置 哨兵模式

1. 安装vim

```
进入三台redis容器: docker exec -it 容器id/容器名称 /bin/bash (固定写法记住就好了)  
更新依赖命令: apt-get  
安装vim命令: apt-get install -y vim
```

2. 三台主机创建哨兵配置文件: sentinel.conf

```
1、进入三台redis容器:  
2、进入容器根目录  
3、创建哨兵配置文件: vim sentinel.conf (三台容器哨兵配置一模一样)  
4. 输入:  
port: 26379 #哨兵端口号 一定要和启动命令映射第二个端口号一致  
daemonize yes #后台启动  
sentinel monitor mymaster 主节点ip 主节点端口 2
```

3. 启动redis哨兵

```
依次三台在sentinel.conf 配置文件同一级目录执行: redis-sentinel sentinel.conf
```

4. 查看是否启动哨兵

```
三台容器依次安装ps命令: apt-get install procps  
依次查看三台容器的是否启动哨兵。
```

8、哨兵模式优缺点

优点:

1. Redis Sentinel 集群部署简单。
2. 对节点进行监控，来完成自动的故障发现与转移，能够解决 Redis 主从模式下的高可用切换问题。
3. 很方便实现 Redis 数据节点的线形扩展，轻松突破 Redis 自身单线程瓶颈，可极大满足 Redis 大容量或高性能的业务需求。
4. 可以实现一套 Sentinel 监控一组 Redis 数据节点或多组数据节点。

缺点:

1. 部署相对 Redis 主从模式要复杂一些，更繁琐。
2. 资源浪费，Redis 数据节点中 slave 节点作为备份节点不提供服务。
3. 与主从相比，哨兵仅解决了手动切换主从节点问题，至于其他的问题，基本上仍然存在。
4. 哨兵的主要问题还是由于中心架构，仅存在一个master节点引起的，写的效率太低。

Redis 集群模式

概述

主从不能解决故障自动恢复问题，哨兵已经可以解决故障自动恢复了，那到底为啥还要集群模式呢？

主从和哨兵都还有另外一些问题没有解决，单个节点的存储能力是有上限，访问能力是有上限的。

Redis Cluster 集群模式具有 **高可用、可扩展性、分布式、容错** 等特性

Redis Cluster采用无中心结构，具备多个节点之间自动进行数据分片的能力，支持节点动态添加与移除，可以在部分节点不可用时进行自动故障转移，确保系统高可用的一种集群化运行模式。

特点

Redis Cluster最大的特点在于可扩展性，多个主节点通过分片机制存储所有数据，即每个主从复制结构单元管理部分key。

因为在主从复制、哨兵模式下，同样具备其他优点。

所以，动态伸缩能力是Redis Cluster最耀眼的特色。

Cluster 集群模式的原理

通过数据分片的方式来进行数据共享问题，同时提供数据复制和故障转移功能。

之前的两种模式数据都是在一个节点上的，单个节点存储是存在上限的。集群模式就是把数据进行分片存储(**哈希槽**)，当一个分片数据达到上限的时候，就分成多个分片。

Redis中只有网络请求模块和数据操作模块是单线程的。而其他的如持久化存储模块、集群支撑模块等是多线程的

哈希槽

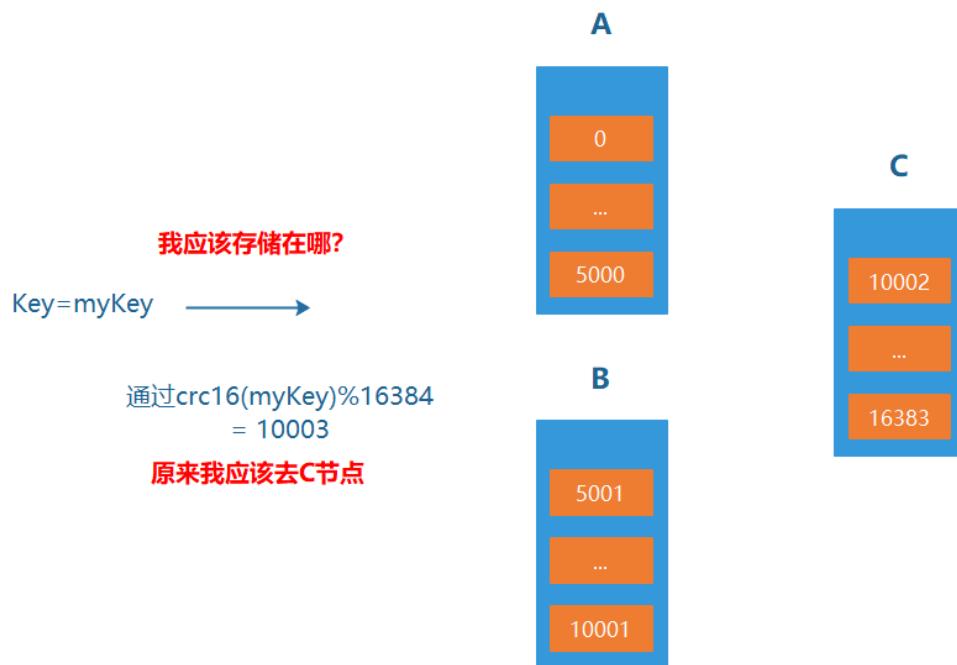
Redis-cluster引入了**哈希槽**的概念。

Redis-cluster中有16384(即 2^{14} 次方) 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。

Cluster中的每个节点负责一部分hash槽 (hash slot) 。

比如集群中存在三个节点，则可能存在的一种分配如下：

- 节点A包含0到5500号哈希槽；
- 节点B包含5501到11000号哈希槽；
- 节点C包含11001 到 16383号哈希槽。



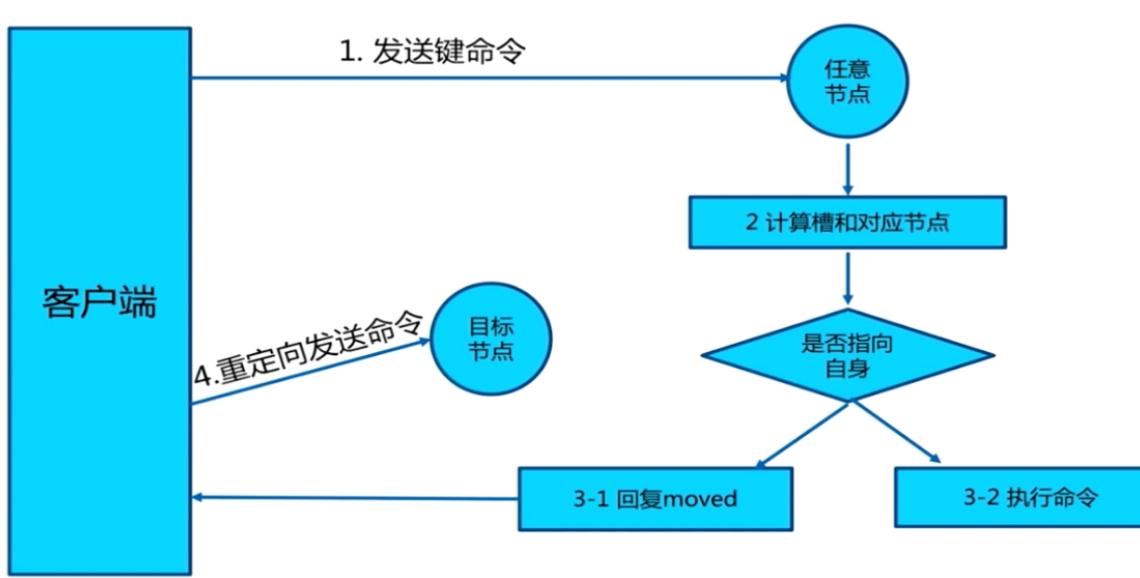
请求重定向

Redis cluster采用去中心化的架构，集群的主节点各自负责一部分槽，客户端如何确定key到底会映射到哪个节点上呢？这就是我们要讲的请求重定向。

在cluster模式下，**节点对请求的处理过程**如下：

- 检查当前key是否存在当前NODE?
 - 通过crc16 (key) /16384计算出slot
 - 查询负责该slot负责的节点，得到节点指针
 - 该指针与自身节点比较
- 若slot不是由自身负责，则返回MOVED重定向
- 若slot由自身负责，且key在slot中，则返回该key对应结果
- 若key不存在此slot中，检查该slot是否正在迁出 (MIGRATING) ?
- 若key正在迁出，返回ASK错误重定向客户端到迁移的目的服务器上
- 若Slot未迁出，检查Slot是否导入中?
- 若Slot导入中且有ASKING标记，则直接操作
- 否则返回MOVED重定向

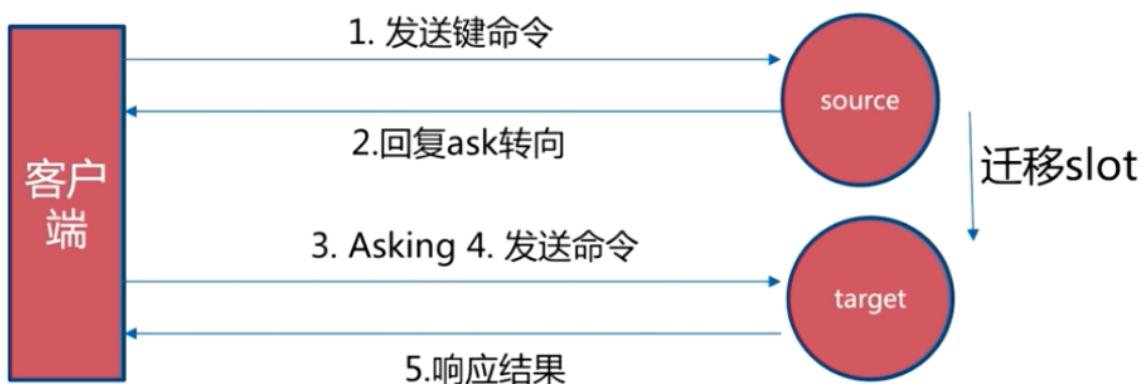
move重定向：



- 槽命中：直接返回结果
- 槽不命中：即当前键命令所请求的键不在当前请求的节点中，则当前节点会向客户端发送一个 Moved 重定向，客户端根据Moved 重定向所包含的内容找到目标节点，再一次发送命令。

ASK 重定向：

Ask重定向发生于集群伸缩时，集群伸缩会导致槽迁移，当我们去源节点访问时，此时数据已经可能已经迁移到了目标节点，使用Ask重定向来解决此种情况



Redis 工作模式小结

主从、哨兵、集群各自架构的优点和缺点对比：https://blog.csdn.net/m0_45406092/article/details/16171758

每种模式都有各自的优缺点，在实际使用场景中要根据业务特点去选择合适的模式。

redis是一个非常常用的中间件，作为一个使用者来说，学习成本一点不高。

如果作为一个很好的中间件去研究的话，还是有很多值得学习和借鉴的地方。比如redis的各种数据结构（动态字符串、跳跃表、集合、字典等）、高效的内存分配(jemalloc)、高效的IO模型等等。

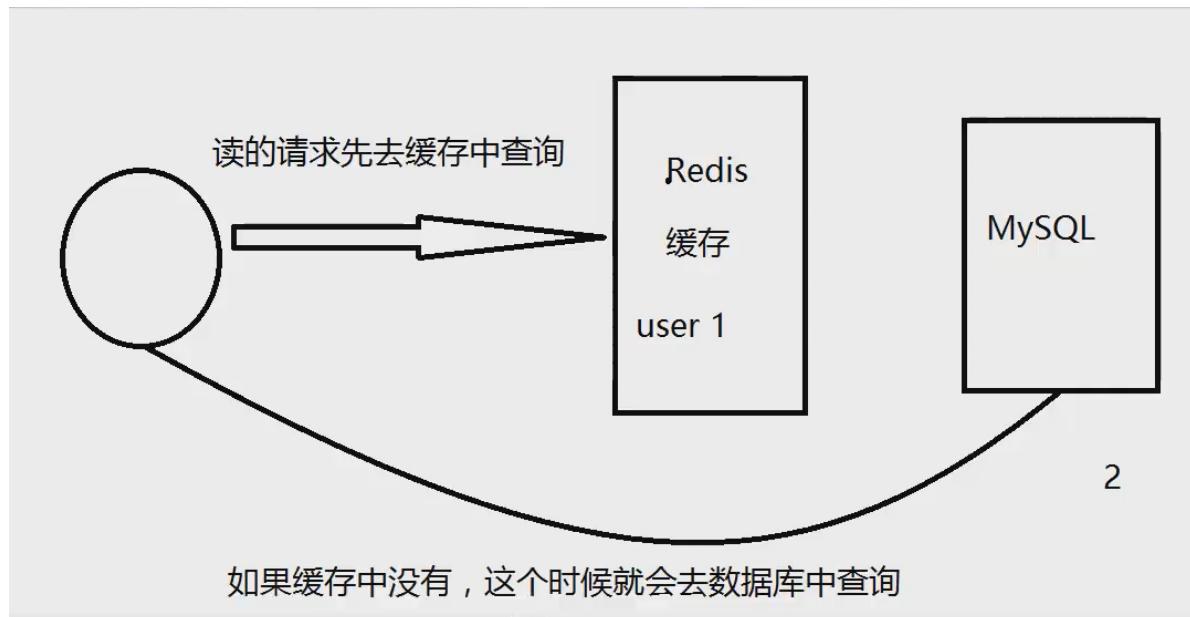
每个点都可以深入研究，在后期设计高并发、高可用系统的时候融入进去。

Redis 缓存穿透和雪崩（服务的高可用问题）

缓存穿透 (查不到)

缓存穿透 缓存中未命中 大量请求到数据库导致宕机

查询一个一定不存在的数据，由于缓存是不命中时需要从数据库查询，查不到数据则不写入缓存，这将导致这个不存在的数据每次请求都要到数据库去查询，造成缓存穿透

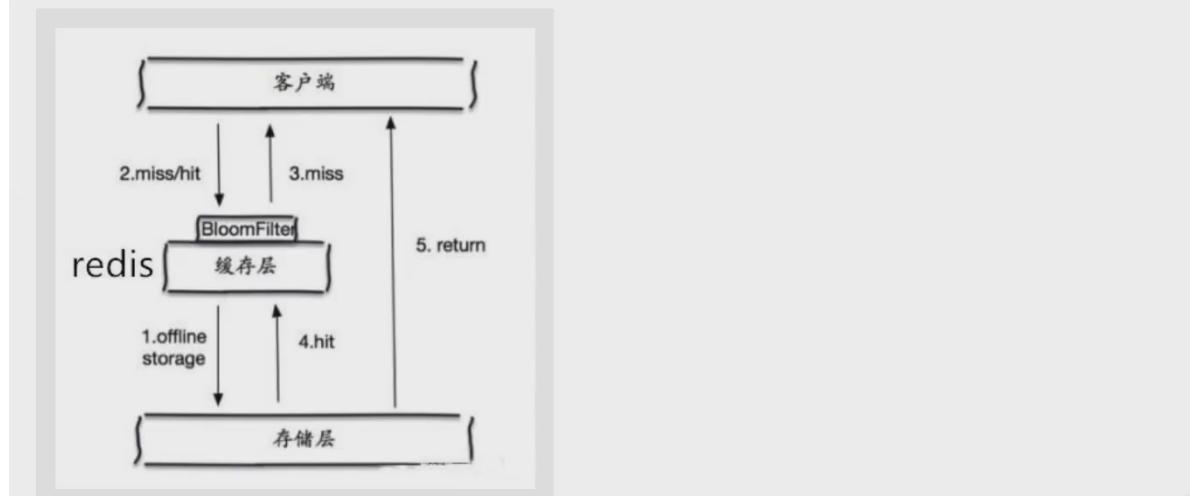


解决方案

1、布隆过滤器

布隆过滤器

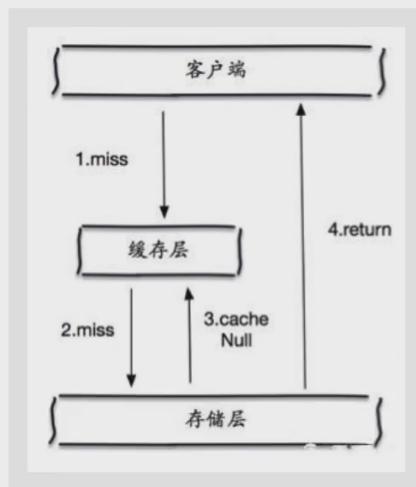
布隆过滤器是一种数据结构，对所有可能查询的参数以hash形式存储，在控制层先进行校验，不符合则丢弃，从而避免了对底层存储系统的查询压力；



2、缓存空对象

缓存空对象

当存储层不命中后，即使返回的空对象也将其缓存起来，同时会设置一个过期时间，之后再访问这个数据将会从缓存中获取，保护了后端数据源；



但是这种方法会存在两个问题：

- 1、如果空值能够被缓存起来，这就意味着缓存需要更多的空间存储更多的键，因为这当中可能会有很多的空值的键；
- 2、即使对空值设置了过期时间，还是会存在缓存层和存储层的数据会有一段时间窗口的不一致，这对于需要保持一致性的业务会有影响。

缓存击穿（缓存过期）

概述

如微博服务器宕机

概述

这里需要注意和缓存击穿的区别，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，就像在一个屏障上凿开了一个洞。

当某个key在过期的瞬间，有大量的请求并发访问，这类数据一般是热点数据，由于缓存过期，会同时访问数据库来查询最新数据，并且回写缓存，会导使数据库瞬间压力过大。

解决方案

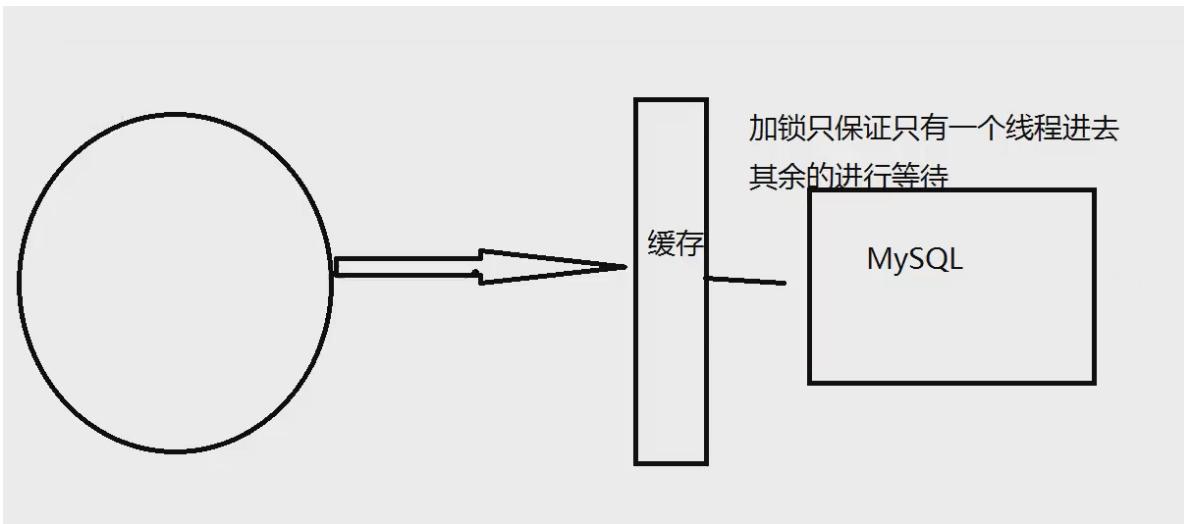
解决方案

设置热点数据永不过期

从缓存层面来看，没有设置过期时间，所以不会出现热点 key 过期后产生的问题。

加互斥锁

分布式锁：使用分布式锁，保证对于每个key同时只有一个线程去查询后端服务，其他线程没有获得分布式锁的权限，因此只需要等待即可。这种方式将高并发的压力转移到了分布式锁，因此对分布式锁的考验很大。



缓存雪崩

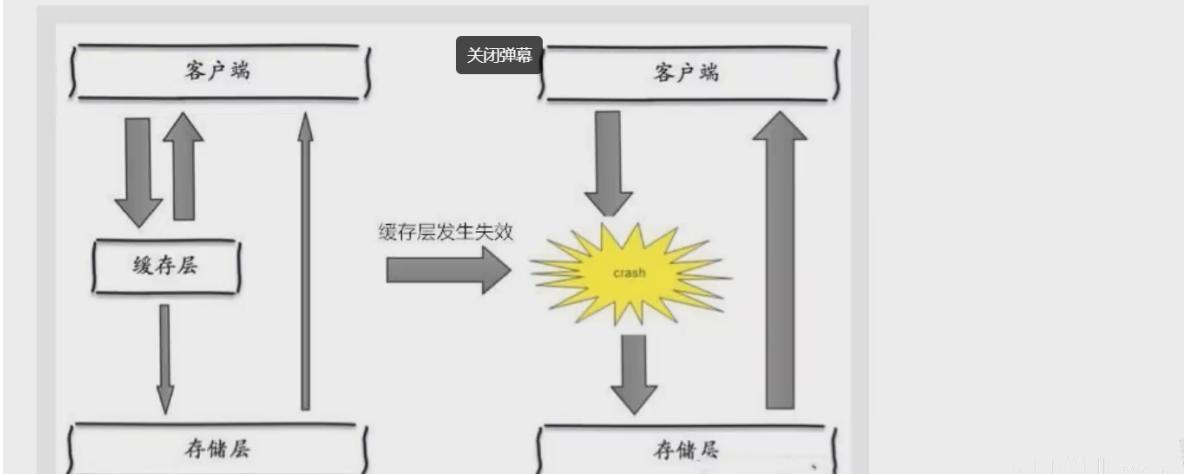
概念

缓存雪崩

概念

缓存雪崩，是指在某一个时间段，缓存集中过期失效。Redis 宕机！

产生雪崩的原因之一，比如在写本文的时候，马上就要到双十二零点，很快就会迎来一波抢购，这波商品时间比较集中的放入了缓存，假设缓存一个小时。那么到了凌晨一点钟的时候，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。于是所有的请求都会达到存储层，存储层的调用量会暴增，造成存储层也会挂掉的情况。



其实集中过期，倒不是非常致命，比较致命的缓存雪崩，是缓存服务器某个节点宕机或断网。因为自然形成的缓存雪崩，一定是在某个时间段集中创建缓存，这个时候，数据库也是可以顶住压力的。无非就是对数据库产生周期性的压力而已。而缓存服务节点的宕机，对数据库服务器造成的影响是不可预知的，很有可能瞬间就把数据库压垮。

解决方案

redis高可用

这个思想的含义是，既然redis有可能挂掉，那我多增设几台redis，这样一台挂掉之后其他的还可以继续工作，其实就是搭建的集群。（异地多活！）

限流降级（在SpringCloud讲解过！）

这个解决方案的思想是，在缓存失效后，通过加锁或者队列来控制读数据库写缓存的线程数量。比如对某个key只允许一个线程查询数据和写缓存，其他线程等待。

数据预热

数据加热的含义就是在正式部署之前，我先把可能的数据先预先访问一遍，这样部分可能大量访问的数据就会加载到缓存中。在即将发生大并发访问前手动触发加载缓存不同的key，设置不同的过期时间，让缓存失效的时间点尽量均匀。

其他

Redis性能如此高的原因，我总结了如下几点：

- 纯内存操作
- 单线程
- 高效的数据结构
- 合理的数据编码
- 其他方面的优化

在 Redis 中，常用的 5 种数据结构和应用场景如下：

- **String**: 缓存、计数器、分布式锁等。
- **List**: 链表、队列、微博关注人时间轴列表等。
- **Hash**: 用户信息、Hash 表等。
- **Set**: 去重、赞、踩、共同好友等。
- **Zset**: 访问量排行榜、点击量排行榜等。

Redis 面试专题

1 什么是Redis？有什么特点

Redis 是一款开源，高性能的 key-value 的非关系型数据库。内存数据库

特点：

- 1) 支持持久化，可以将内存中的数据持久化到磁盘，重启可以再次从磁盘中加载使用；
- 2) 支持多种数据结构；
- 3) 支持数据的备份：主从模式的备份；支持集群
- 4) 高性能，读速度达 11 万次/秒，写速度达到 8.1 万次/秒
- 5) 支持事务

2 Redis 的数据类型

一共 8 种

5 种基本数据类型：String、Hash、List、Set、Zset

3 种特殊类型：geospatial、hyperloglog、bitmap

3 Redis 和 Memcache 的区别？

- 1) Memcache 数据都存储在内存中，断电即失，数据不能超过内存大小；而 Redis 的数据可以持久化到硬盘。
- 2) Memcache 只支持简单的字符串，Redis 有丰富的数据结构；
- 3) 底层实现方式不一样，Redis 自行构建了 VM 机制，速度更快。

4 Redis 是单线程的？

Redis 将数据放在内存中，单线程执行效率最高，多线程执行反而需要进行 CPU 上下文切换，这是个耗时操作，单线程效率最高。

Redis6.0 已经支持多线程，但默认关闭，需要手动开启。

5 Redis的持久化

Redis 提供了两种持久化机制：RDB 和 AOF

RDB 持久化机制指的是，用数据集快照的方式记录 Redis 数据库的所有键值对，在某个时间点写入一个临时文件，持久化结束后，用这个临时文件替换上次持久化的文件，达到数据恢复的目的。

优点：

- 1) 只有一个文件 dump.rdb 方便持久化；
- 2) 容灾性好，一个文件可以保存到安全的磁盘；
- 3) 性能最大化，Redis 会单独创建（fork）一个子进程进行持久化，主进程不进行任何 IO 操作，保证了性能；
- 4) 在数据较多时，比 AOF 的启动效率高。

缺点：

最后一次持久化的数据可能会丢失。

AOF 持久化，是以独立日志的方式记录每次写命令，并在 Redis 重启时重新执行 AOF 文件中的命令以达到恢复数据的目的。AOF 主要解决数据持久化的实时性。

优点：

- 1) 数据安全，配置 appendfsync 属性，可以选择不同的同步策略；
- 2) 自动修复功能，redis-check-aof 工具可以解决数据一致性问题；

缺点：

- 1) AOF 文件比 RDB 文件大，且恢复速度慢；
- 2) 数据多时，效率低于 RDB。

AOF 如何防止文件过大？AOF 重写，只保留最后一次的修改记录。

6 Redis 的主从复制

主从复制指的是将一台 Redis 服务器的数据复制到其他 Redis 服务器，前者称之为 **主节点**，后者称之为 **从节点**。

主从复制的作用：

- 1) 数据冗余：主从复制实现了数据的热备份；
- 2) 故障修复：当主节点出现故障后，从节点还可以提供服务，实现快速的故障修复。
- 3) 负载均衡：在主从复制的基础上，配合读写分离，可以由主节点提供写操作，从节点提供读操作，实现负载均衡，提高并发量；
- 4) 高可用的基石：主从复制是哨兵模式的基础。

复制原理：

从节点启动成功连接到主节点后，会发送一个 sync 的同步命令。主节点接收到命令之后，启动后台的存盘进程，收集所有修改数据库的命令，在后台执行完毕后将整个数据文件传送到从节点，完成一次完全同步。

全量复制：从节点在接收到了数据文件后，将其存盘文件加载到内存中；

增量复制：主节点继续将新收集到修改命令传递给从节点，完成同步

7 说说哨兵模式

哨兵模式是为了解决手动切换主节点的问题。Redis 提供了哨兵的命令，哨兵是一个**独立的进程**。哨兵能够后台监控主节点是否故障，如果故障需要将从节点选举为主节点。

其原理是哨兵通过发送命令，等待 Redis 服务器的响应，从而监控多个 Redis 节点。

当只有一个哨兵时，还是可能会出现问题的，比如哨兵自己挂掉。为此，可以使用多哨兵模式，多个哨兵之间相互监控。当主节点宕机了，哨兵1先检测到这个结果，系统并不会马上进行 failover 【故障转移】的过程。仅仅是哨兵1认为主节点不可用的现象称之为**主观下线**。当其余的哨兵也检测到主节点不可用之后，哨兵之间会进行一次投票选举从节点中的一个作为新的主节点，这个过程称之为**客观下线**。

哨兵模式的优点：

- 1) 基于主从复制，高可用；
- 2) 主从可以切换，进行故障转移，系统可用性好；
- 3) 哨兵模式是主从模式的升级版，手动到自动，更加健壮。

哨兵模式的缺点：

- 1) 不方便在线扩容；
- 2) 实现哨兵模式需要很多的配置。

8 缓存穿透、缓存击穿和缓存雪崩

缓存穿透：

概念：用户需要查询一个数据，缓存中没有，也就是没有命中，于是向数据库中发起请求，发现也没有。当用户很多的时候，缓存都没有命中，于是都去请求数据库，这给数据库造成很大的压力。

解决方案：

- 布隆过滤器：是一种数据结构，对所有可能查询的参数以 hash 方式存储，先在控制层进行校验，不符合则丢弃，避免了过多访问数据库。
- 缓存空对象：当存储层没有命中时，即使返回空对象也将其缓存起来。（意味着更多的空间存储，即使设置了过期时间，缓存和数据库还是有段时间数据不一致。）

缓存击穿：

概念：当一个 key 非常热点时，在不断扛高并发，集中对这个热点数据进行访问，当这个 key 失效的瞬间，请求直接到达数据库，给数据库瞬间的高压力。

解决方案：

- 设置热点数据永不过期
- 加分布式锁(setnx、setex)：保证每个 key 同时只有一个线程去查询后端服务。

缓存雪崩：

概念：某个时间段，缓存集中失效

解决方案：

- 增加 Redis 集群的数量
- 缓存过期时间的时候，错峰设置
- 限流降级：在缓存失效后，通过加锁和队列来控制数据库写缓存的线程数量
- 数据预热：正式部署之前，将数据预先访问一遍，让缓存失效的时间尽量均匀

9 Redis 的使用场景

- 1) 会话缓存：如单点登录，使用 Redis 模拟 session，SSO 系统生成一个 token，将用户信息存到 Redis 中，并设置过期时间；
- 2) 全页缓存
- 3) 作为消息队列平台
- 4) 排行榜和计数器
- 5) 发布/订阅：比如聊天系统
- 6) 热点数据：比如ES中搜索的热词

10 Redis 缓存如何保持一致性

读数据的时候首先去 Redis 中读取，没有读到再去 MySQL 中读取，读取到数据更新到 Redis 中作为下一次的缓存。

写数据的时候会产生数据不一致的问题。无论是先写入 Redis 再写入 MySQL 中，还是先写入 MySQL 再写入 Redis 中，这两步操作都不能保证原子性，所以会出现 Redis 和 MySQL 中数据不一致的问题。

无论采取何种方式都不能保证强一致性，如果对 Redis 中的数据设置了过期时间，能够保证最终一致性，对架构的优化只能降低发生的概率，不能从根本上避免不一致性。

更新缓存的两种方式：删除失效缓存、更新缓存

更新缓存和数据库有两种顺序：先数据库后缓存(防止缓存到期了直接访问数据库造成击穿)、先缓存后数据库

两者组合，分为四种更新策略。

11 集群

redis cluster 分为 16384 个槽，最多可支持 1000 个节点，数据存放时，根据 $\text{CRC16}(\text{key}) \% 16384$ 取模，得到对应的槽，存放到槽对应的节点上。

取数据时，会计算槽，进行请求重定向，如果在当前节点，直接返回数据，不在 move 到对应节点，获取数据