# Redis简介

## 基本介绍

1. **海量用户访问、高并发，导致系统访问瓶颈？**

**原因分析：**

1. 关系型数据库最终是要与磁盘交互，进行IO操作
2. 数据关系复杂，扩展性差，不便于大规模集群

**解决思路：**

1. 降低磁盘IO次数，越低越好 --内存存储
2. 去除数据间关系，越简单越好 --不存储关系，只存数据
3. **Nosql**

NoSQL：即Not-OnlySQL(泛指非关系型的数据库)，作为关系型数据库的补充。

作用：应对基于海量用户和海量数据前提下的数据处理问题。

特征（相较于关系型数据库）：

* 可扩容，可伸缩
* 大数据量下高性能
* 灵活的数据模型
* 高可用

1. **常见Nosql数据库：**

* redis
* memcache
* HBase
* MongoDB

## 解决方案（电商场景）

1. 商品基本信息：名称、价格、厂商等：

使用Mysql关系型数据库存储

1. 商品附加信息：描述、详情、评论

使用MongoDB（文档存储）

1. 图片信息

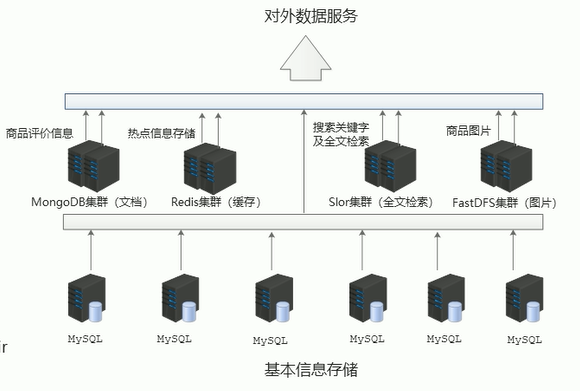
使用分布式文件系统存储

1. 搜索关键字

使用ES、Lucene、solr等搜索引擎

1. 热点信息：高频、波段性、内容不确定性，可以是1-4任意一种

使用Redis、memcache、tair内存存储。



## Redis

**概念：**Redis(Remote Dictionary server) 使用C语言开发的一个开源的高性能键值对数据库

**特征：**

1. 数据间没有必然联系
2. 内部采用单线程机制进行工作
3. 高性能。官方提供测试数据，50个并发执行100000个请求，读速度是110000次/s,写速度是81000次/s。
4. 多数据类型支持

* 字符串 string
* 列表类型 list
* 散列类型 hash
* 集合类型 set
* 有序集合类型 sorted\_set

1. 持久化支持，可以进行数据灾难恢复（不是主推的东西）

## Redis的应用

* 为热点数据（热点数据一定是由普通数据变为热点数据的）加速查询（主要场景），如热点商品、热点新闻、热点资讯、推广类等高访问量信息等。
* 任务队列，如秒杀、抢票、购票
* 即时信息查询，如各位排行榜、各类网站访问统计、公交到站信息、在线人数信息（聊天室、网站）、设备信号灯
* 时效性信息控制，如验证码控制、投票控制等
* 分布式数据共享，如分布式集群架构中的session分离
* 消息队列
* 分布式锁

# Redis下载与安装

## windows

* 下载地址：

<https://github.com/microsoftarchive/redis>

* 核心文件简述

redis-server.exe 服务器启动命令

redis-cli.exe 命令行客户端

redis.windows.conf redis核心配置文件

redis-benchmark.exe 性能测试工具

redis-check-aof.exe AOF文件修复工具

redis-check-dump.exe RDB文件检查工具（快照持久化文件）

* 启动服务端

如果出现错误，可以用控制台执行redis-server.exe就能看见错误，最好是带配置文件启动，例如：D:\tools\Redis-x64-3.2.100>redis-server.exe redis.windows.conf

* 启动客户端

双击redis-cli.exe

如果出现：

|  |
| --- |
| Creating Server TCP listening socket 127.0.0.1:6379: bind: No error |

则执行：

|  |
| --- |
| redis-cli.exe  shutdown  exit  redis-server redis.windows.conf |

## centos7

1. 安装gcc依赖

由于 redis 是用 C 语言开发，安装之前必先确认是否安装 gcc 环境（gcc -v），如果没有安装，执行以下命令进行安装

**[root@localhost local]# yum install -y gcc**

1. 下载并解压安装包

下载地址中，可以选版本

wget <http://download.redis.io/releases/redis-5.0.3.tar.gz>

tar -zxvf redis-5.0.3.tar.gz

1. cd切换到redis解压目录下，执行编译

**[root@localhost local]# cd redis-5.0.3**

**[root@localhost redis-5.0.3]# make**

1. 安装并指定安装目录

**[root@localhost redis-5.0.3]# make install PREFIX=/usr/local/redis**

1. 默认配置启动服务（前台启动，不建议）

**[root@localhost redis-5.0.3]# cd /usr/local/redis/bin/**

**[root@localhost bin]# ./redis-server**

1. 配置文件启动服务（后台启动，建议）

从 redis 的源码目录中复制 redis.conf 到 redis 的安装目录

**cp /usr/local/download/redis-5.0.3/redis.conf /usr/local/redis/bin/**

修改 redis.conf 文件，把 daemonize no 改为 daemonize yes

**[root@localhost bin]# vi redis.conf**

使用配置文件、后台启动

**[root@localhost redis-5.0.3]# cd /usr/local/redis/bin/**

**[root@localhost bin]# ./redis-server redis.conf**

1. 设置开机启动（注意创建redis.service,需要改变权限）

* 添加开机启动服务：注意：ExecStart配置成自己的路径

**[root@localhost bin]# vi /etc/systemd/system/redis.service**

|  |
| --- |
| [Unit] Description=redis-server After=network.target  [Service] Type=forking ExecStart=/usr/local/redis/bin/redis-server /usr/local/redis/bin/redis.conf PrivateTmp=true  [Install] WantedBy=multi-user.target |

* 设置开机启动

**[root@localhost bin]# systemctl daemon-reload**

**[root@localhost bin]# systemctl start redis.service**

**[root@localhost bin]# systemctl enable redis.service**

* 创建 redis 命令软链接

**[root@localhost ~]# ln -s /usr/local/redis/bin/redis-cli /usr/bin/redis**

* **查看服务列表（当启动失败时，查看服务是否添加成功）**

**systemctl list-unit-files --type=service**

1. 服务操作命令

systemctl start redis.service   #启动redis服务

systemctl stop redis.service   #停止redis服务

systemctl restart redis.service   #重新启动服务

systemctl status redis.service   #查看服务当前状态

systemctl enable redis.service   #设置开机自启动

systemctl disable redis.service   #停止开机自启动

# Redis 的基本操作

## 使用命令行思考

任何命令行，初学者可按照以下思路去了解：

* 功能性命令
* 清除屏幕信息
* 帮助信息查阅
* 退出指令

## 常用命令

1. 信息添加

set key value

1. 信息查询

get key

1. 清除屏幕信息

clear

1. 退出客户端

quit exit <ESC>

1. 帮助命令

help 命令名称

help @组名

help 查询说明：

To get help about Redis commands type:

"help @<group>" to get a list of commands in <group>

"help <command>" for help on <command>

"help <tab>" to get a list of possible help topics

"quit" to exit

# Redis数据类型

## 数据存储类型介绍

* **Redis存储的业务场景**

1. 原始业务功能设计：

秒杀、618活动、双11活动、排队购票

1. 运营平台监控到的突发高频访问数据

突发时政要闻，被强势关注围观

1. 高频、复杂的统计数据

直播的在线人数、投票排行榜

1. 附加功能

单服务器升级集群（存储分布式信息）、Session 管理、Token 管理

* Redis 数据类型（5种常用）

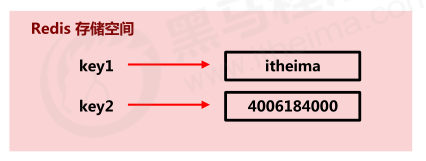
|  |  |
| --- | --- |
| Redis数据类型 | 类比Java数据类型 |
| string | string |
| hash | HashMap |
| list | LinkedList |
| set | HashSet |
| sorted\_set | TreeSet |

## String

### 基本操作

1. **简介**

* 存储的数据：单个数据，最简单的数据存储类型，也是最常用的数据存储类型
* 存储数据的格式：一个存储空间保存一个数据
* 存储内容：通常使用字符串，如果字符串以整数的形式展示，可以作为数字操作使用



1. **基本操作**

* 添加/修改数据

set key value

* 获取数据

get key

* 删除数据

del key

* 添加/修改多个数据（m代表Multiple）

mset key1 value1 key2 value2

* 获取多个数据（m代表Multiple）

mget key1 key2

* 获取数据字符个数（字符串长度）

strlen key

* 追加信息到原始信息后部（如果原始信息存在就追加，否则新建）

append key value

1. **单数据操作与多数据操作的选择：**

如果需要执行数据较多，可以选择多数据操作，并且也可以将数据切分



### 作为mysql全局主键（数值操作）

* **业务场景**

大型企业级应用中，分表操作是基本操作，使用多张表存储同类型数据，但是对应的主键 id 必须保证统一性，不能重复。Oracle 数据库具有 sequence 设定，可以解决该问题，但是 MySQL数据库并不具有类似的机

制，那么如何解决？

* **解决方案**

1. 设置数值数据增加指定范围的值

|  |
| --- |
| incr key  incrby key increment  incrbyfloat key increment |

1. 设置数值数据减少指定范围的值

|  |
| --- |
| decr key  decrby key increment |

* **string 作为数值操作**
* string在redis内部存储默认就是一个字符串，当遇到增减类操作incr，decr时会转成数值型进行计算。
* redis所有的操作都是原子性的，采用单线程处理所有业务，命令是一个一个执行的，因此无需考虑并发带来的数据影响。
* 注意：按数值进行操作的数据，如果原始数据不能转成数值，或超越了redis 数值上限范围，将报错。9223372036854775807（java中long型数据最大值，Long.MAX\_VALUE）
* **Tips 1：**
* redis用于控制数据库表主键id，为数据库表主键提供生成策略，保障数据库表的主键唯一性
* 此方案适用于所有数据库，且支持数据库集群

### token超时（设置超时时间）

* **业务场景**

1. “最强女生”启动海选投票，只能通过微信投票，每个微信号每 4 小时只能投1票。
2. 电商商家开启热门商品推荐，热门商品不能一直处于热门期，每种商品热门期维持3天，3天后自动取消热门。
3. 新闻网站会出现热点新闻，热点新闻最大的特征是时效性，如何自动控制热点新闻的时效性

* **解决方案**

设置数据具有指定的生命周期

|  |
| --- |
| setex key seconds value  psetex key milliseconds value |

* **Tips 2**

redis 控制数据的生命周期，通过数据是否失效控制业务行为，适用于所有具有时效性限定控制的操作

### 类型数据操作的注意事项

1. 数据操作不成功的反馈与数据正常操作之间的差异

表示运行结果是否成功：(integer) 0 → false 失败 (integer) 1 → true 成功

表示运行结果值：(integer) 3 → 3 3个 (integer) 1 → 1 1个

1. 数据未获取到

（nil）等同于null

1. 数据最大存储量

512MB

1. 数值计算最大范围（java中的long的最大值）

9223372036854775807 ~ -9223372036854775807

### key 的设置约定

* **key 的设置约定**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 表名 | 主键名 | 主键值 | 字段名 |
| eg1 | order | id | 29437595 | name |
| eg2 | equip | id | 390472345 | type |
| eg3 | news | id | 202004150 | title |

* **业务场景**

主页高频访问信息显示控制，例如新浪微博大V主页显示粉丝数与微博数量

* **解决方案一：按照数据表的字段维度存储**

在redis中为大V用户设定用户信息，以用户主键和属性值作为key，后台设定定时刷新策略即可,**好处可以直接计算**

eg: user:id:3506728370:fans → 12210947

eg: user:id:3506728370:blogs → 6164

eg: user:id:3506728370:focuss → 83

* **解决方案二：存储json数据**

在redis中以json格式存储大V用户信息，定时刷新（也可以使用hash类型），**好处根据id获取某人的全量数据**

eg: user:id:3506728370 →

{"id":3506728370,"name":"春晚","fans":12210862,"blogs":6164, "focus":83}

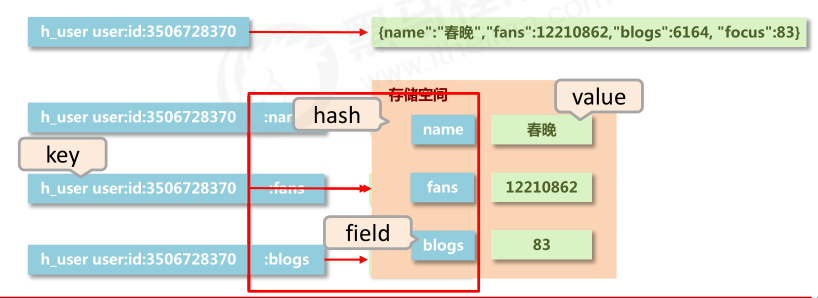
* **Tips 3**

redis应用于各种结构型和非结构型高热度数据访问加速

## hash类型

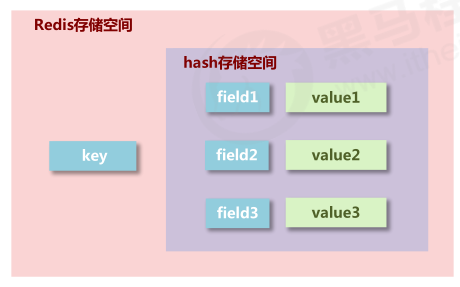
### hash类型出现背景

对象类数据的存储如果具有较频繁的更新需求操作会显得笨重



### hash 类型

* 新的存储需求：对一系列存储的数据进行编组，方便管理，典型应用存储对象信息
* 需要的存储结构：一个存储空间保存多个键值对数据
* hash类型：底层使用哈希表结构实现数据存储
* hash存储结构优化：如果field数量较少，存储结构优化为类数组结构、如果field数量较多，存储结构使用HashMap结构



### hash 类型数据的基本操作

1. 添加/修改数据：hset key field value
2. 获取数据：hget key field hgetall key
3. 删除数据：hdel key field1 field2…
4. 添加/修改多个数据：hmset key field1 value1 field2 value2 …
5. 获取多个数据：hmget key field1 field2…
6. 获取哈希表中字段的数量：hlen key
7. 获取哈希表中是否存在指定的字段：hexists key field

### hash类型数据扩展操作

1. 获取哈希表中所有的字段名或字段值

hkeys key

hvals key

1. 设置指定字段的数值数据增加指定范围的值

hincrby key field increment

hincrbyfloat key field increment

1. 如果有就不添加，如果没有则添加

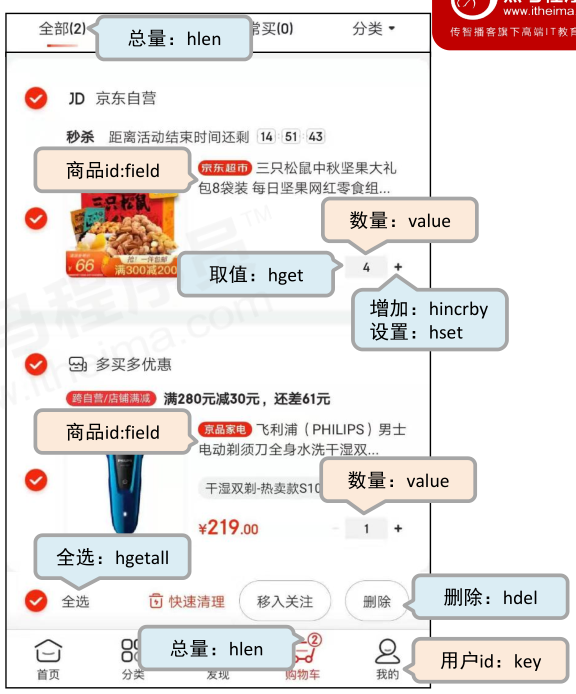
hsetnx key field value

### hash 类型数据操作的注意事项

1. hash类型下的value只能存储字符串，不允许存储其他数据类型，不存在嵌套现象。如果数据未获取到，对应的值为（nil）
2. 每个 hash 可以存储 2 32 - 1 个键值对
3. hash类型十分贴近对象的数据存储形式，并且可以灵活添加删除对象属性。但hash设计初衷不是为了存储大量对象而设计的，切记不可滥用，更不可以将hash作为对象列表使用
4. hgetall 操作可以获取全部属性，如果内部field过多，遍历整体数据效率就很会低，有可能成为数据访问瓶颈

### hash类型应用场景（购物车）

1. **业务场景**



1. **业务分析**

* 仅分析购物车的redis存储模型：添加、浏览、更改数量、删除、清空
* 购物车于数据库间持久化同步（不讨论）
* 购物车于订单间关系（不讨论）

提交购物车：读取数据生成订单

商家临时价格调整：隶属于订单级别

* 未登录用户购物车信息存储（不讨论）：cookie存储

1. **业务数据模型要求**

存储用户信息、用户订购产品的数量、产品的基本信息（描述、图片、单价）

1. **业务关键点设计**
2. 应用启动时不能全部加载到redis,因此只有第一次访问时需要将商品信息加入到redis中。
3. 商品信息是通用信息，所以可将产品信息单独存入到hash中，并且可以按照商品的类目存储，减少hash检索的压力
4. 经以上分析，购物车的在redis的存储模型为：

users:id:001 productId1:nums productId2:nums

可以根据productId获取产品信息

1. 购物车中的产品价格调动问题，使用hsetnx key field value（如果有就不添加，如果没有则添加）

### hash类型应用场景（抢购）

1. **业务场景**

双11活动日，销售手机充值卡的商家对移动、联通、电信的30元、50元、100元商品推出抢购活动，每种商品抢购上限1000张

1. **解决方案**
   * + 以商家id作为key
     + 将参与抢购的商品id作为field
     + 将参与抢购的商品数量作为对应的value
     + 抢购时使用降值的方式控制产品数量
     + 实际业务中还有超卖等实际问题，这里不做讨论
2. **Tips 5**

redis 应用于抢购，限购类、限量发放优惠卷、激活码等业务的数据**存储设计(不要将业务判断放入redis中)**

1. **string存储对象（json）与hash存储对象**

使用string对象存储json字符串，在取出数据时更能保证数据的完整性

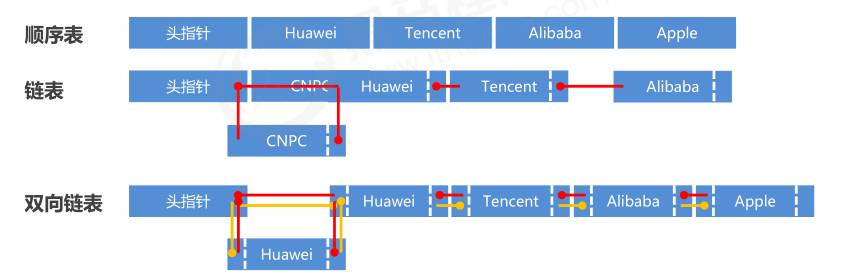
hash存储对象，方便对字段进行更新操作。

## list类型（双向链表）

由于字符串和hash不能存储大量的数据，因此产生了list类型，并且还有存放的顺序

### 基本概念

* 数据存储需求：存储多个数据，并对数据进入存储空间的**顺序**进行区分
* 需要的存储结构：一个存储空间保存多个数据，且通过数据可以体现进入顺序
* list类型：保存多个数据，底层使用**双向链表存储结构**实现





### 基本操作

* 添加/修改数据

lpush key value1 [value2] ……

rpush key value1 [value2] ……

* 获取数据

lrange key start stop

(lrange list1 0 -1)全部取出

lindex key index

（lindex list1 -1）获取最后一位元素

llen key

* 获取并移除数据

lpop key

rpop key

### 阻塞操作

规定时间内获取并移除数据

blpop key1 key2 timeout

brpop key1 [key2] timeout

brpoplpush source destination timeout

### 删除指定值

* **业务场景**

微信朋友圈点赞，要求按照点赞顺序显示点赞好友信息

如果取消点赞，移除对应好友信息



* **解决方案**

移除指定数据

lrem key count value

* **Tips 6**

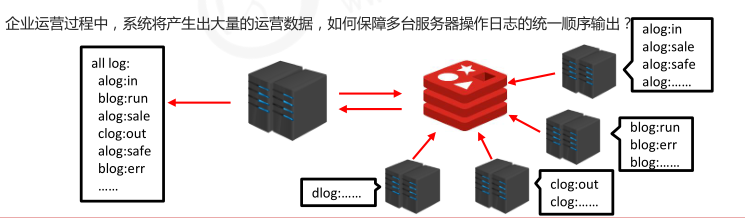
redis 应用于具有操作先后顺序的数据控制

### 类型数据操作注意事项

* list中保存的数据都是string类型的，数据总容量是有限的，最多2 32 - 1 个元素 (4294967295)。
* list具有索引的概念，但是操作数据时通常以队列的形式进行入队出队操作，或以栈的形式进行入栈出栈操作
* 获取全部数据操作结束索引设置为-1
* list可以对数据进行分页操作，通常第一页的信息来自于list，第2页及更多的信息通过数据库的形式加载

### 应用场景

1. twitter、新浪微博、腾讯微博中个人用户的关注列表需要按照用户的关注顺序进行展示，粉丝列表需要将最近关注的粉丝列在前面
2. 新闻、资讯类网站如何将最新的新闻或资讯按照发生的时间顺序展示？
3. 企业运营过程中，系统将产生出大量的运营数据，如何保障多台服务器操作日志的统一顺序输出？



1. 解决方案

* 依赖list的数据具有顺序的特征对信息进行管理
* 使用队列模型解决多路信息汇总合并的问题
* 使用栈模型解决最新消息的问题

## set类型

### set类型出现背景

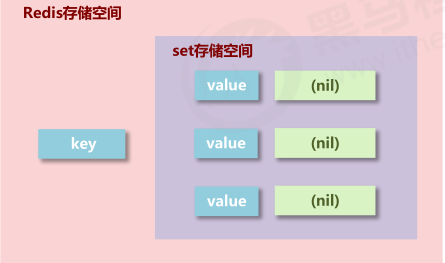
1. 新的存储需求：存储大量的数据，在查询方面提供更高的效率
2. 需要的存储结构：能够保存大量的数据，高效的内部存储机制，便于查询

list虽然能存储大量数据，但是由于是双向链表的结果，因此查找数据效

率不是很高。

hash查询效率更高，但是不能存储大量的数据，因此，我们在hash的基础上将value设置为nil,只保留key值存储数据，这样就形成的一种数据结构set

1. set类型：与hash存储结构完全相同，仅存储键，不存储值（nil），并且值是不允许重复的



### 基本操作

* 添加数据

sadd key member1 [member2]

* 获取全部数据

smembers key

* 删除数据

srem key member1 [member2]

* 获取集合数据总量

scard key

* 判断集合中是否包含指定数据

sismember key member

### 扩展操作（随机获取）

* **业务场景**

每位用户首次使用今日头条时会设置3项爱好的内容，但是后期为了增加用户的活跃度、兴趣点，必须让用户对其他信息类别逐渐产生兴趣，增加客户留存度，如何实现？

* **业务分析**

1. 系统分析出各个分类的最新或最热点信息条目并组织成set集合
2. 随机挑选其中部分信息
3. 配合用户关注信息分类中的热点信息组织成展示的全信息集合

* **解决方案**

1. 随机获取集合中指定数量的数据

srandmember key [count]

1. 随机获取集合中的某个数据并将该数据移出集合

spop key [count]

* **Tips 8**

redis 应用于随机推荐类信息检索，例如热点歌单推荐，热点新闻推荐，热卖旅游线路，应用APP推荐，

大V推荐等

其实还可以利用redis进行复杂的任务分发。例如1000个任务。存入到redis中，我们可以随机获取一定数量任务。通过注册中心获取微服务列表。进行策略分发。（这样做的好处是，使用redis，保证了高可用）

### 扩展操作（交、并、差）

* **业务场景**

1. 脉脉为了促进用户间的交流，保障业务成单率的提升，需要让每位用户拥有大量的好友，事实上职场新人不具有更多的职场好友，如何快速为用户积累更多的好友？（好友推荐）
2. 新浪微博为了增加用户热度，提高用户留存性，需要微博用户在关注更多的人，以此获得更多的信息或热门话题，如何提高用户关注他人的总量？
3. QQ新用户入网年龄越来越低，这些用户的朋友圈交际圈非常小，往往集中在一所学校甚至一个班级中，如何帮助用户快速积累好友用户带来更多的活跃度？
4. 微信公众号是微信信息流通的渠道之一，增加用户关注的公众号成为提高用户活跃度的一种方式，如何帮助用户积累更多关注的公众号？
5. 美团外卖为了提升成单量，必须帮助用户挖掘美食需求，如何推荐给用户最适合自己的美食？

* **解决方案**

1. 求两个集合的交、并、差集

sinter key1 [key2]

sunion key1 [key2]

sdiff key1 [key2]

1. 求两个集合的交、并、差集并存储到指定集合中

sinterstore destination key1 [key2]

sunionstore destination key1 [key2]

sdiffstore destination key1 [key2]

1. 将指定数据从原始集合中移动到目标集合中

smove source destination member

* **Tips 9：**

1. redis 应用于同类信息的关联搜索，二度关联搜索，深度关联搜索
2. 显示共同关注（一度）
3. 显示共同好友（一度）
4. 由用户A出发，获取到好友用户B的好友信息列表（一度）
5. 由用户A出发，获取到好友用户B的购物清单列表（二度）
6. 由用户A出发，获取到好友用户B的游戏充值列表（二度）

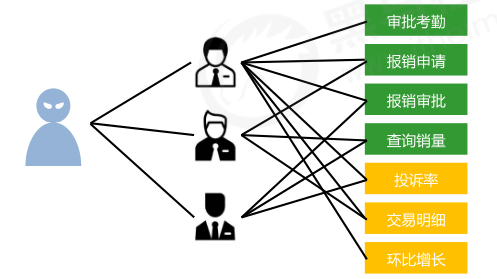
### 注意事项

1. set 类型不允许数据重复，如果添加的数据在 set 中已经存在，将只保留一份
2. set 虽然与hash的存储结构相同，但是无法启用hash中存储值的空间

### set 类型应用场景（权限校验）

* 业务场景

集团公司共具有12000名员工，内部OA系统中具有700多个角色，3000多个业务操作，23000多种数据，每位员工具有一个或多个角色，如何快速进行业务操作的权限校验？



### set 类型应用场景（去重统计）

* **业务场景**

公司对旗下新的网站做推广，统计网站的PV（访问量）,UV（独立访客）,IP（独立IP）。

1. PV：网站被访问次数，可通过刷新页面提高访问量
2. UV：网站被不同用户访问的次数，可通过cookie统计访问量，相同用户切换IP地址，UV不变
3. IP：网站被不同IP地址访问的总次数，可通过IP地址统计访问量，相同IP不同用户访问，IP不变

* **解决方案**

1. 利用set集合的数据去重特征，记录各种访问数据
2. 建立string类型数据，利用incr统计日访问量（PV）
3. 建立set模型，记录不同cookie数量（UV）
4. 建立set模型，记录不同IP数量（IP）

* **Tips 11**

redis 应用于同类型数据的快速去重

### set 类型应用场景（黑白名单）

* **业务场景**

1. 黑名单

资讯类信息类网站追求高访问量，但是由于其信息的价值，往往容易被不法分子利用，通过爬虫技术，快速获取信息，个别特种行业网站信息通过爬虫获取分析后，可以转换成商业机密进行出售。例如第三方火车票、机票、酒店刷票代购软件，电商刷评论、刷好评。同时爬虫带来的伪流量也会给经营者带来错觉，产生错误的决策，有效避免网站被爬虫反复爬取成为每个网站都要考虑的基本问题。在基于技术层面区分出爬虫用户后，需要将此类用户进行有效的屏蔽，这就是**黑名单**的典型应用。ps:不是说爬虫一定做摧毁性的工作，有些小型网站需要爬虫为其带来一些流量。

1. 白名单

对于安全性更高的应用访问，仅仅靠黑名单是不能解决安全问题的，此时需要设定可访问的用户群体，依赖**白名单**做更为苛刻的访问验证。

* **解决方案**

1. 基于经营战略设定问题用户发现、鉴别规则
2. 周期性更新满足规则的用户黑名单，加入set集合
3. 用户行为信息达到后与黑名单进行比对，确认行为去向
4. 黑名单过滤IP地址：应用于开放游客访问权限的信息源
5. 黑名单过滤设备信息：应用于限定访问设备的信息源
6. 黑名单过滤用户：应用于基于访问权限的信息源

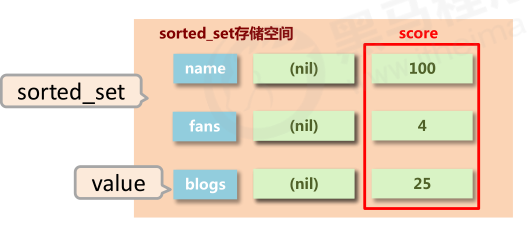
* **Tips 12：**

redis 应用于基于黑名单与白名单设定的服务控制

## sorted\_set 类型

### sorted\_set出现背景

* 新的存储需求：数据排序有利于数据的有效展示，需要提供一种可以根据自身特征进行排序的方式
* 需要的存储结构：新的存储模型，可以保存可排序的数据
* sorted\_set类型：在set的存储结构基础上添加可排序字段



### 基本操作

* 添加数据

zadd key score1 member1 [score2 member2]

* 获取全部数据

zrange key start stop [withscores]

zrevrange key start stop [withscores]

* 删除数据

zrem key member [member …]

* 按条件获取数据

zrangebyscore key min max [withscores] [limit]

zrevrangebyscore key max min [WITHSCORES]

* 条件删除数据

zremrangebyrank key start stop

zremrangebyscore key min max

注意：

1. min与max用于限定搜索查询的条件
2. start与stop用于限定查询范围，作用于索引，表示开始和结束索引
3. offset与count用于限定查询范围，作用于查询结果，表示开始位置和数据总量

* 获取集合数据总量

zcard key

zcount key min max

* 集合交、并操作

zinterstore destination numkeys key [key ...] [WEIGHTS weight] [AGGREGATE SUM|MIN|MAX]

zunionstore destination numkeys key [key ...] [WEIGHTS weight] [AGGREGATE SUM|MIN|MAX]

### 扩展操作（获取排序索引、排序值）

* **业务场景**

1. 票选广东十大杰出青年，各类综艺选秀海选投票
2. 各类资源网站TOP10（电影，歌曲，文档，电商，游戏等）
3. 聊天室活跃度统计
4. 游戏好友亲密度

* **业务分析**

为所有参与排名的资源建立排序依据

* **解决方案**

获取数据对应的索引（排名）

zrank key member

zrevrank key member

score值获取与修改

zscore key member

zincrby key increment member

### 注意事项

1. score保存的数据存储空间是64位，如果是整数范围是-9007199254740992~9007199254740992
2. score保存的数据也可以是一个双精度的double值，基于双精度浮点数的特征，可能会丢失精度，使用时候要慎重
3. sorted\_set 底层存储还是基于set结构的，因此数据不能重复，如果重复添加相同的数据，score值将被反复覆盖，保留最后一次修改的结果

### 应用场景（任务时效性管理）

* **业务场景**

基础服务+增值服务类网站会设定各位会员的试用，让用户充分体验会员优势。例如观影试用VIP、游戏VIP体验、云盘下载体验VIP、数据查看体验VIP。当VIP体验到期后，如果有效管理此类信息。即便对于正式VIP用户也存在对应的管理方式。

网站会定期开启投票、讨论，限时进行，逾期作废。如何有效管理此类过期信息。

* **解决方案**

1. 对于基于时间线限定的任务处理，将处理时间记录为score值，利用排序功能区分处理的先后顺序
2. 记录下一个要处理的时间，当到期后处理对应任务，移除redis中的记录，并记录下一个要处理的时间
3. 当新任务加入时，判定并更新当前下一个要处理的任务时间
4. 为提升sorted\_set的性能，通常将任务根据特征存储成若干个sorted\_set。例如1小时内，1天内，周内，月内，季内，年度等，操作时逐级提升，将即将操作的若干个任务纳入到1小时内处理的队列中

* **获取当前系统时间**

time

* **Tips 14**

redis 应用于定时任务执行顺序管理或任务过期管理

### 应用场景（权重的任务管理）

* **业务场景**

任务/消息权重设定应用：当任务或者消息待处理，形成了任务队列或消息队列时，对于高优先级的任务要保障对其优先处理，如何实现任务权重管理。

* **解决方案**

1. 对于带有权重的任务，优先处理权重高的任务，采用score记录权重即可
2. 多条件任务权重设定：

如果权重条件过多时，需要对排序score值进行处理，保障score值能够兼容2条件或者多条件，例如外贸订单优先于国内订单，总裁订单优先于员工订单，经理订单优先于员工订单

因score长度受限，需要对数据进行截断处理，尤其是时间设置为小时或分钟级即可（折算后）

先设定订单类别，后设定订单发起角色类别，整体score长度必须是统一的，不足位补0。第一排序规则首位不得是0

* 例如外贸101，国内102，经理004，员工008。
* 员工下的外贸单score值为101008（优先）
* 经理下的国内单score值为102004
* **Tips 15**

redis 应用于即时任务/消息队列执行管理

## 数据类型实践案例

### 案例一（按次结算服务控制）

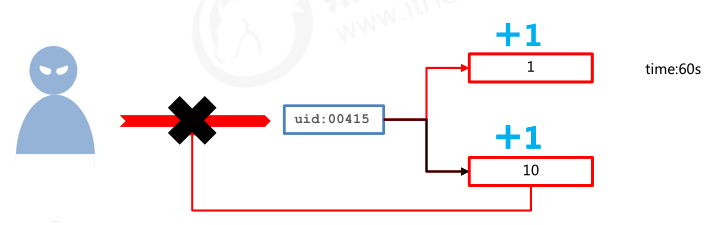
* **业务场景**

人工智能领域的语义识别与自动对话将是未来服务业机器人应答呼叫体系中的重要技术，百度自研用户评价语义识别服务，免费开放给企业试用，同时训练百度自己的模型。现对试用用户的使用行为进行限速，限制

每个用户每分钟最多发起10次调用

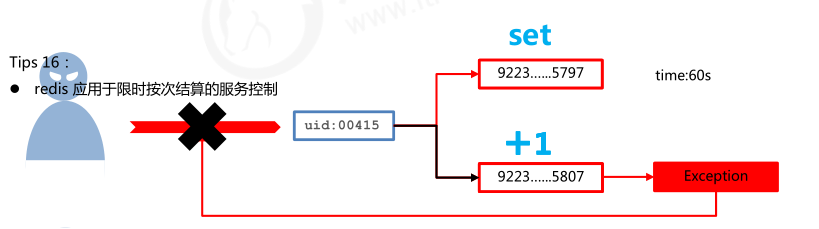
* **解决方案**

1. 设计计数器，记录调用次数，用于控制业务执行次数。以用户id作为key，使用次数作为value
2. 在调用前获取次数，判断是否超过限定次数：不超过次数的情况下，每次调用计数+1；业务调用失败，计数-1
3. 为计数器设置生命周期为指定周期，例如1秒/分钟，自动清空周期内使用次数



* **方案改良**

1. 取消最大值的判定，利用incr操作超过最大值抛出异常的形式替代每次判断是否大于最大值
2. 判断是否为nil：如果是，设置为Max-次数。如果不是，计数+1。业务调用失败，计数-1
3. 遇到异常即+操作超过上限，视为使用达到上限



* **Tips 16**

redis 应用于限时按次结算的服务控制

### 案例二（微信接受顺序消息控制）

* **业务场景**

使用微信的过程中，当微信接收消息后，会默认将最近接收的消息置顶，当多个好友及关注的订阅号同时发送消息时，该排序会不停的进行交替。同时还可以将重要的会话设置为置顶。一旦用户离线后，再次打开微信时，消息该按照什么样的顺序显示？

* **业务分析**



* **解决方案**

1. 依赖list的数据具有顺序的特征对消息进行管理，将list结构作为栈使用
2. 对置顶与普通会话分别创建独立的list分别管理
3. 当某个list中接收到用户消息后，将消息发送方的id从list的一侧加入list（此处设定左侧）
4. 多个相同id发出的消息反复入栈会出现问题，在入栈之前无论是否具有当前id对应的消息，先删除对应id
5. 推送消息时先推送置顶会话list，再推送普通会话list，推送完成的list清除所有数据
6. 消息的数量，也就是微信用户对话数量采用计数器的思想另行记录，伴随list操作同步更新

* **Tips 17**

redis 应用于基于时间顺序的数据操作，而不关注具体时间

### 案例总结

Tips 1：redis用于控制数据库表主键id，为数据库表主键提供生成策略，保障数据库表的主键唯一性

Tips 2：redis 控制数据的生命周期，通过数据是否失效控制业务行为，适用于所有具有时效性限定控制的操作

Tips 3：redis应用于各种结构型和非结构型高热度数据访问加速

Tips 4：redis 应用于购物车数据存储设计

Tips 5：redis 应用于抢购，限购类、限量发放优惠卷、激活码等业务的数据存储设计

Tips 6：redis 应用于具有操作先后顺序的数据控制

Tips 7：redis 应用于最新消息展示

Tips 8：redis 应用于随机推荐类信息检索，例如热点歌单推荐，热点新闻推荐，热卖旅游线路，应用APP推荐，大V推荐等

Tips 9：redis 应用于同类信息的关联搜索，二度关联搜索，深度关联搜索

Tips 10：redis 应用于同类型不重复数据的合并、取交集操作

Tips 11：redis 应用于同类型数据的快速去重

Tips 12：redis 应用于基于黑名单与白名单设定的服务控制

Tips 13：redis 应用于计数器组合排序功能对应的排名

Tips 14：redis 应用于定时任务执行顺序管理或任务过期管理

Tips 15：redis 应用于及时任务/消息队列执行管理

Tips 16：redis 应用于按次结算的服务控制

Tips 17：redis 应用于基于时间顺序的数据操作，而不关注具体时间

# Redis通用指令

## key通用指令

* **通用指令分类**

1. 对于key自身状态的相关操作，例如：删除，判定存在，获取类型等
2. 对于key有效性控制相关操作，例如：有效期设定，判定是否有效，有效状态的切换等
3. 对于key快速查询操作，例如：按指定策略查询key

* **通用指令**

1. 删除指定key

del key

1. 获取key是否存在

exists key

1. 获取key的类型

type key

1. 为指定key设置有效期

expire key seconds

pexpire key milliseconds

expireat key timestamp

pexpireat key milliseconds-timestamp

1. 获取key的有效时间

ttl key（-1:没有时效控制；-2：key不存在；大于0，表示剩余时间）

pttl key

1. 切换key从时效性转换为永久性

persist key

1. 查询key

keys pattern

查询模式规则:

|  |
| --- |
| keys \* 查询所有  keys it\* 查询所有以it开头  keys \*heima 查询所有以heima结尾  keys ??heima 查询所有前面两个字符任意，后面以heima结尾  keys user:? 查询所有以user:开头，最后一个字符任意  keys u[st]er:1 查询所有以u开头，以er:1结尾，中间包含一个字母，s或t |

1. 为key改名

rename key newkey(如果重名就覆盖)

renamenx key newkey（如果newkey不存在才能修改成功）

1. 对所有key排序

sort

SORT key [BY pattern] [LIMIT offset count] [GET pattern [GET pattern ...]] [ASC|DESC] [ALPHA] [STORE destination]

1. 其他key通用操作

help @generic

## 数据库通用指令

* **出现背景**

1. key是由程序员定义的，容易重复
2. redis在使用过程中，伴随着操作数据量的增加，会出现大量的数据以及对应的key
3. 数据不区分种类、类别混杂在一起，极易出现重复或冲突

* **解决方案**

1. redis为每个服务提供有16个数据库，编号从0到15
2. 每个数据库之间的数据相互独立

* **DB操作**

1. 切换数据库

select index

1. 其他操作

quit

ping

echo message

1. 数据所在库的位置移动

move key db (当前库必须存在key,并且目标库，key不存在)

1. 数据库清除

dbsize (数据库key存储个数)

flushdb(清除当前数据库)

flushall(清除所有数据库)

# Jedis

jedis是java连接redis的服务。其他的工具还有SpringData Redis等

## HelloWorld（Jedis版）

* **jar包导入**

下载地址：

<https://mvnrepository.com/artifact/redis.clients/jedis>

基于maven:

|  |
| --- |
| <dependency>  <groupId>redis.clients</groupId>  <artifactId>jedis</artifactId>  <version>2.9.0</version>  </dependency> |

* **客户端连接redis**

1. 连接redis

|  |
| --- |
| Jedis jedis = new Jedis("localhost", 6379); |

1. 操作redis

|  |
| --- |
| jedis.set("name", "itheima");  jedis.get("name"); |

1. 关闭redis连接

|  |
| --- |
| jedis.close(); |

* **API文档**

<http://xetorthio.github.io/jedis/>

## Jedis读写redis数据

Jedis操作数据命令与redis客户端命令一致。

jedis.lpush(key, value…);

jedis.lrange(key, start, end);

jedis.hset(key, filed, value);

jedis.hgetAll(key);

jedis.hget(key, field);

## 服务调用次数控制实现

* 案例背景

人工智能领域的语义识别与自动对话将是未来服务业机器人应答呼叫体系中的重要技术，百度自研用户评价语义识别服务，免费开放给企业试用，同时训练百度自己的模型。现对试用用户的使用行为进行限速，限制每个用户每分钟最多发起10次调用

* 案例要求

1. 设定A、B、C三个用户
2. A用户限制10次/分调用，B用户限制30次/分调用，C用户不限制

* 需求分析

1. 设定一个服务方法，用于模拟实际业务调用的服务，内部采用打印模拟调用
2. 在业务调用前服务调用控制单元，内部使用redis进行控制，参照之前的方案
3. 对调用超限使用异常进行控制，异常处理设定为打印提示信息
4. 主程序启动3个线程，分别表示3种不同用户的调用

* 实现代码逻辑

1. 初始化用户访问次数
2. 编写服务控制业务逻辑（超过次数，则提示;否则调用业务处理）
3. 编写业务处理（只是打印一句话）
4. 模拟多线程客户端调用



## Jedis简易工具类开发



# 基于Linux环境安装Redis

## 下载与安装

详见2.2 contos7

## 配置文件（基本配置）redis.conf

1. 从原始配置文件中整理拷贝

**cat /usr/local/download/redis-5.0.3/redis.conf | grep -v "#" | grep -v "^$" > /usr/local/redis/bin/redis-6379.conf**

1. 配置选项

daemonize yes

以守护进程方式启动，使用本启动方式，redis将以服务的形式存在，日志将不再打印到命令窗口中

port 6\*\*\*

设定当前服务启动端口号

dir “/自定义目录/redis/data“

设定当前服务文件保存位置，包含日志文件、持久化文件（后面详细讲解）等

logfile "6\*\*\*.log“

设定日志文件名，便于查阅

## 创建日志存放路径

在配置文件中配置dir 和 logfile 值，用于存放日志路径

## 启动命令

* 默认配置启动

redis-server

redis-server –-port 6379

redis-server –-port 6380 ……

* 指定配置文件启动

redis-server redis.conf

redis-server redis-6379.conf

redis-server redis-6380.conf ……

redis-server conf/redis-6379.conf

redis-server config/redis-6380.conf ……

* 默认连接

redis-cli

* 连接指定服务器

redis-cli -h 127.0.0.1

redis-cli –p 6379

redis-cli -h 127.0.0.1 –p 6379

# 持久化（RDB）

## 持久化简介

* **持久化意义**

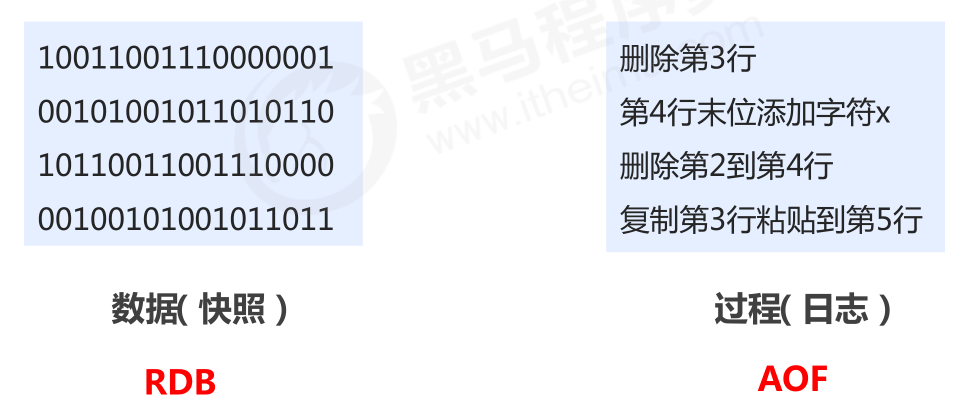
防止数据的意外丢失，确保数据安全性

* **释义**

利用永久性存储介质将数据进行保存，在特定的时间将保存的数据进行恢复的工作机制称为持久化。

* **持久化保存什么（RDB和AOF）**

1. 将当前数据状态进行保存，**快照形式**，存储数据结果，存储格式简单，关注点在数据
2. 将数据的操作过程进行保存，**日志形式**，存储操作过程，存储格式复杂，关注点在数据的操作过程



## RDB（数据快照）持久化

* **释义：**

redis操作者（用户）即时（随时进行）主动保存数据（持久化）

* **命令(主动发起)**

save

* **作用**

手动执行一次保存操作

* save指令相关配置
* dbfilename dump.rdb

说明：设置本地数据库文件名，默认值为 dump.rdb

经验：通常设置为dump-**端口号**.rdb

* dir

说明：设置存储.rdb文件的路径

经验：通常设置成存储空间较大的目录中，目录名称data

* rdbcompression yes

说明：设置存储至本地数据库时是否压缩数据，默认为 yes，采用 LZF 压缩

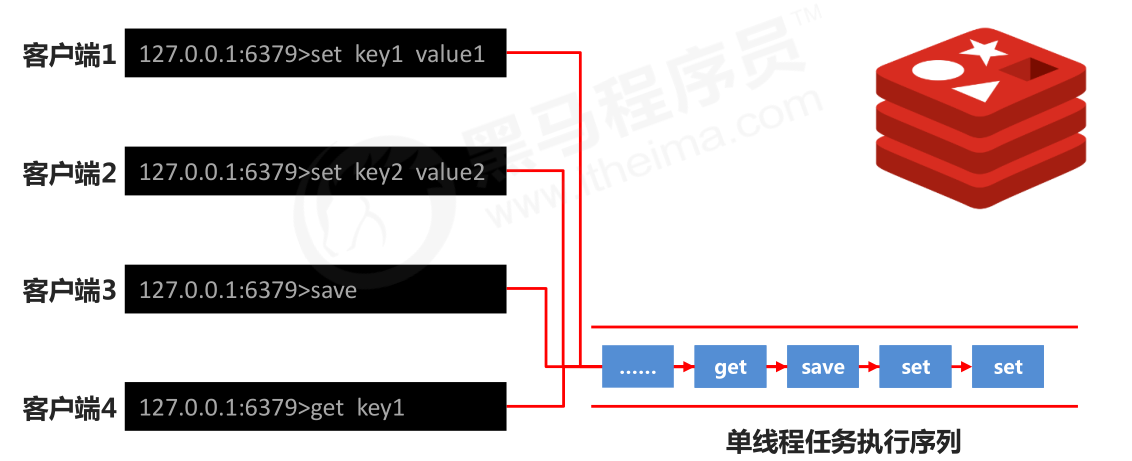
经验：通常默认为开启状态，如果设置为no，可以节省 CPU 运行时间，但会使存储的文件变大（巨大）

* rdbchecksum yes

说明：设置是否进行RDB文件格式校验，该校验过程在写文件和读文件过程均进行

经验：通常默认为开启状态，如果设置为no，可以节约读写性过程约10%时间消耗，但是存储一定的数据损坏风险

## save指令工作原理



注意：save指令的执行会阻塞当前Redis服务器，直到当前RDB过程完成为止，有可能会造成**长时间阻塞**，**线上环境不建议使用**。

## bgsave指令

1. 背景：

由于save命令可能会造成长时间阻塞，因此采用bgsave(后台保存)

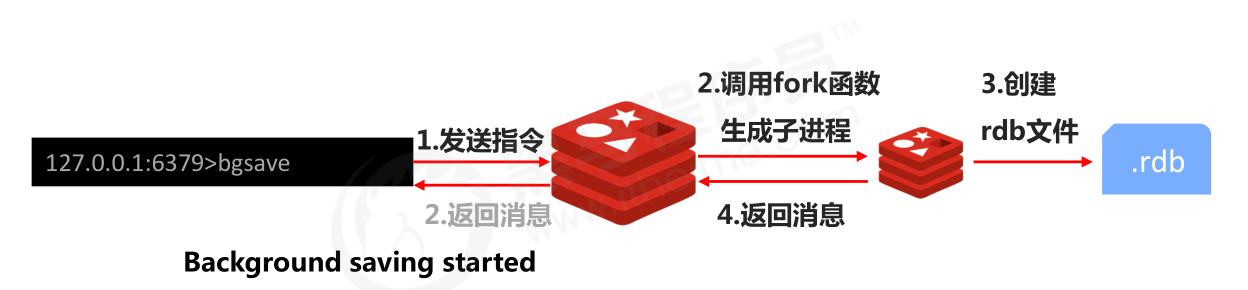
1. 命令

bgsave

1. 作用

手动启动后台保存操作，但不是立即执行

redis操作者（用户）发起指令合理的时间（执行）保存数据



1. 注意

bgsave命令是针对save阻塞问题做的优化。Redis内部所有涉及到RDB操作都采用bgsave的方式，save命令可以放弃使用。

1. 相关配置

stop-writes-on-bgsave-error yes

说明：后台存储过程中如果出现错误现象，是否停止保存操作

经验：通常默认为开启状态

## 服务端自动保存

1. 背景

反复执行保存指令，忘记了怎么办？不知道数据产生了多少变化，何时保存？

1. 释义

redis服务器发起指令（基于条件），当满足条件就保存数据

1. 配置

save second changes

1. 作用

满足限定时间范围内key的变化数量达到指定数量即进行持久化

1. 参数解释

second：监控时间范围

changes：监控key的变化量

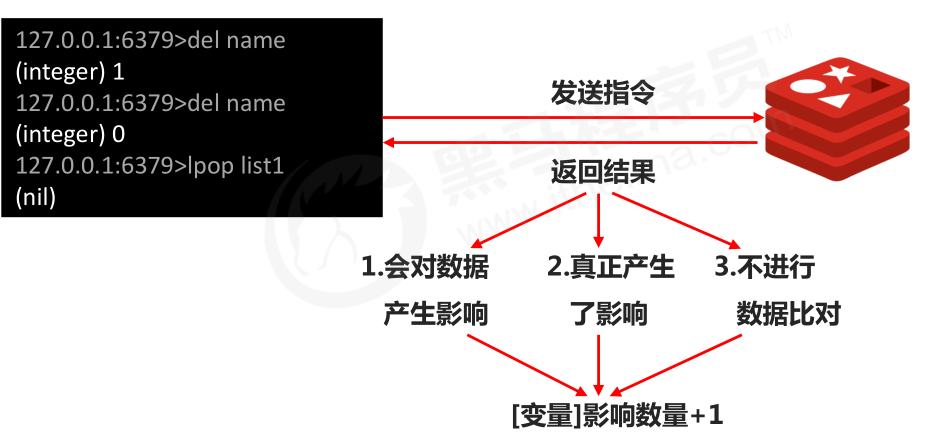
1. 位置

在conf文件中进行配置

1. 范例

|  |
| --- |
| save 900 1  save 300 10  save 60 10000 |

1. 配置原理



1. 注意

save配置要根据实际业务情况进行设置，频度过高或过低都会出现性能问题，结果可能是灾难性的

save配置中对于second与changes设置通常具有互补对应关系，尽量不要设置成包含性关系

save配置启动后执行的是bgsave操作

## RDB总结

* **RDB三种启动方式对比**

|  |  |  |
| --- | --- | --- |
| 方式 | save指令 | bgsave指令 |
| 读写 | 同步 | 异步 |
| 阻塞客户端指令 | 是 | 否 |
| 额外内存消耗 | 否 | 是 |
| 启动新进程 | 否 | 是 |

* **rdb特殊启动形式**
* 全量复制

在主从复制中详细讲解

* 服务器运行过程中重启

debug reload

* 关闭服务器时指定保存数据

shutdown save

默认情况下执行shutdown命令时，自动执行bgsave(如果没有开启AOF持久化功能)

* **RDB优点**

1. RDB是一个紧凑压缩的二进制文件，存储效率较高
2. RDB内部存储的是redis在某个时间点的数据快照，非常适合用于数据备份，全量复制等场景
3. RDB恢复数据的速度要比AOF快很多
4. 应用：服务器中每X小时执行bgsave备份，并将RDB文件拷贝到远程机器中，用于灾难恢复。

* **Rdb缺点**

1. RDB方式无论是执行指令还是利用配置，无法做到实时持久化，具有较大的可能性丢失数据
2. bgsave指令每次运行要执行fork操作创建子进程，要牺牲掉一些性能
3. Redis的众多版本中未进行RDB文件格式的版本统一，有可能出现各版本服务之间数据格式无法兼容现象

# 持久化（AOF）

## AOF出现背景

* **我们开始使用RDB进行持久化的时候，发现如下问题：**

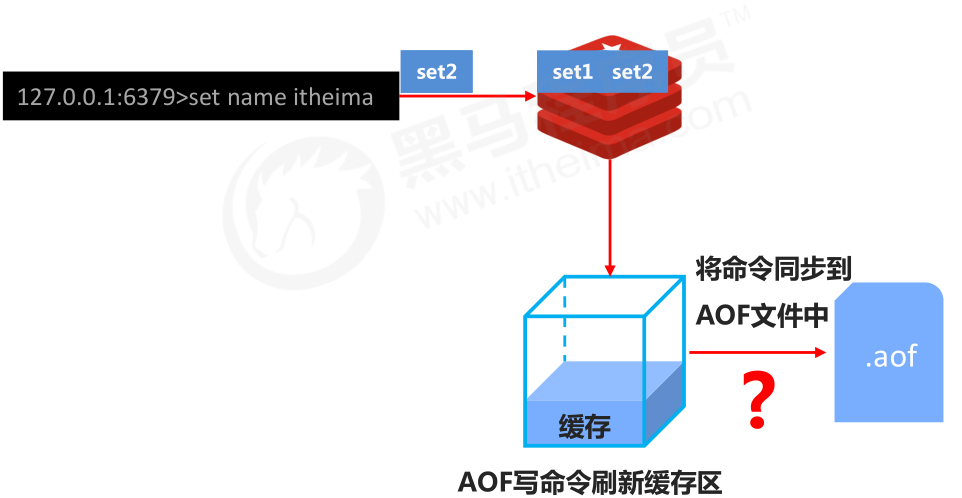
1. 储数据量较大，效率较低。基于快照思想，每次读写都是全部数据，当数据量巨大时，效率非常低
2. 大数据量下的IO性能较低
3. 基于fork创建子进程，内存产生额外消耗
4. 宕机带来的数据丢失风险

* **针对于RDB弊端，提出如下解决思路**

1. 不写全数据，仅记录部分数据（变化）
2. 降低区分数据是否改变的难度，改记录数据为记录操作过程
3. 对所有操作均进行记录，排除丢失数据的风险

## AOF概念

1. AOF(append only file)持久化：以独立日志的方式记录每次写命令，重启时再重新执行AOF文件中命令达到恢复数据的目的。与RDB相比可以简单描述为改记录数据为记录数据产生的过程
2. AOF的主要作用是解决了数据持久化的实时性，目前已经是Redis持久化的主流方式
3. AOF写数据过程：



## AOF三种策略与配置

1. 三种策略

* always(每次）

每次写入操作均同步到AOF文件中，数据零误差，**性能较低，不建议使用**。

* everysec（每秒）

每秒将缓冲区中的指令同步到AOF文件中，数据准确性较高，性能较高，**建议使用，也是默认配置在系统突然宕机的情况下丢失1秒内的数据**

* no（系统控制）

由操作系统控制每次同步到AOF文件的周期，**整体过程不可控**

1. AOF配置

是否开启AOF持久化功能，默认为不开启状态

**appendonly yes|no**

保存策略

**appendfsync always|everysec|no**

AOF持久化文件名，默认文件名未appendonly.aof，建议配置为appendonly-端口号.aof

**appendfilename filename**

AOF持久化文件保存路径，与RDB持久化文件保持一致即可

**dir**

## AOF重写

* **AOF重写**

随着命令不断写入AOF，文件会越来越大，为了解决这个问题，Redis引入了AOF重写机制压缩文件体积。AOF文件重写是将Redis进程内的数据转化为写命令同步到新AOF文件的过程。简单说就是将对同一个数据的若干个条命令执行结果转化成最终结果数据对应的指令进行记录。

* **AOF重写作用**

1. 降低磁盘占用量，提高磁盘利用率
2. 提高持久化效率，降低持久化写时间，提高IO性能
3. 降低数据恢复用时，提高数据恢复效率

* **AOF重写规则**

1. 进程内已超时的数据不再写入文件
2. 忽略无效指令，重写时使用进程内数据直接生成，这样新的AOF文件只保留最终数据的写入命令。如del key1、 hdel key2、srem key3、set key4 111、set key4 222等
3. 对同一数据的多条写命令合并为一条命令

如lpush list1 a、lpush list1 b、 lpush list1 c 可以转化为：lpush list1 a b c。

为防止数据量过大造成客户端缓冲区溢出，对list、set、hash、zset等类型，每条指令最多写入64个元素

* **AOF重写方式**

1. 手动重写

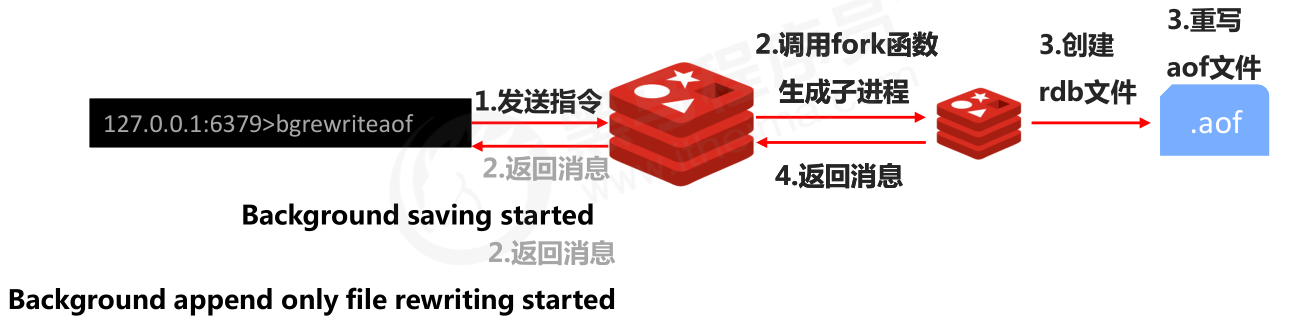
**bgrewriteaof**

1. 自动重写

**auto-aof-rewrite-min-size size**

**auto-aof-rewrite-percentage percentage**

* **bgrewriteaof指令工作原理**



## AOF自动重写配置

* **自动重写触发条件设置**

|  |
| --- |
| auto-aof-rewrite-min-size size  auto-aof-rewrite-percentage percent |

* **自动重写触发比对参数（ 运行指令info Persistence获取具体信息 ）**

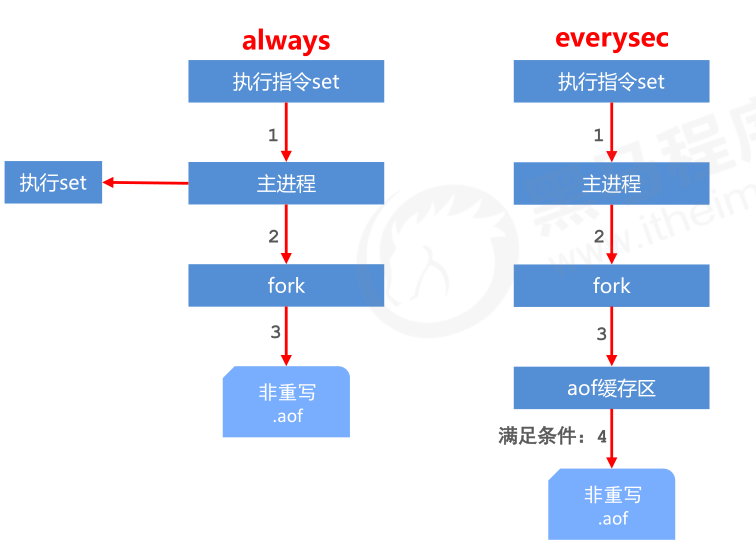
|  |
| --- |
| aof\_current\_size  aof\_base\_size |

* **自动重写触发条件**

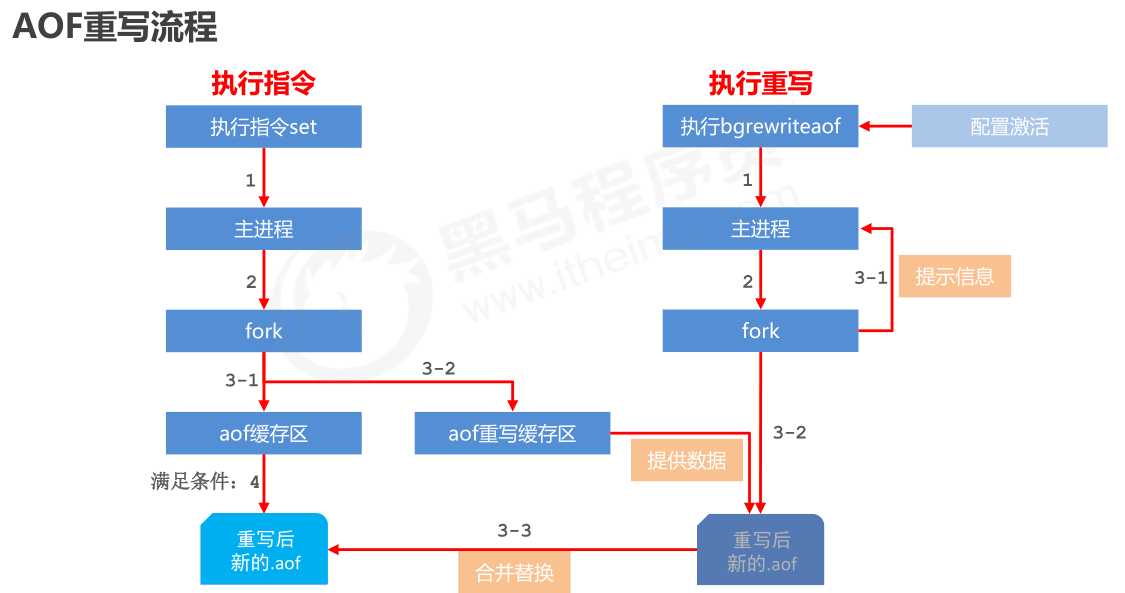
|  |
| --- |
| aof\_current\_size > auto-aof-rewrite-min-size  (aof\_base\_size - aof\_current\_size) / aof\_base\_size > auto-aof-rewrite-percentage |

## AOF非重写、重写工作流程

* 非重写



* 重写



AOF缓冲区同步文件策略，由参数appendfsync控制

系统调用write和fsync说明：

write操作会触发延迟写（delayed write）机制，Linux在内核提供页缓冲区用来提高硬盘IO性能。write操作在写入系统缓冲区后直接返回。同步硬盘操作依赖于系统调度机制，列如：缓冲区页空间写满或达到特定时间周期。同步文件之前，如果此时系统故障宕机，缓冲区内数据将丢失。

fsync针对单个文件操作（比如AOF文件），做强制硬盘同步，fsync将阻塞知道写入硬盘完成后返回，保证了数据持久化。除了write、fsync、Linx还提供了sync、fdatasync操作，具体API说明参见：

# 持久化RDB与AOF对比与选择

## RDB VS AOF

|  |  |  |
| --- | --- | --- |
| 持久化方式 | RDB | AOF |
| 占用存储空间 | 小（数据级：压缩） | 大（指令级：重写） |
| 存储速度 | 慢 | 快 |
| 恢复速度 | 快 | 慢 |
| 数据安全性 | 会丢失数据 | 依据策略决定 |
| 资源消耗 | 高/重量级 | 低/轻量级 |
| 启动优先级 | 低 | 高 |

## RDB与AOF的选择

1. 对数据非常敏感，建议使用默认的AOF持久化方案

AOF持久化策略使用everysecond，每秒钟fsync一次。该策略redis仍可以保持很好的处理性能，当出现问题时，最多丢失0-1秒内的数据。

注意：由于AOF文件存储体积较大，且恢复速度较慢

1. 数据呈现阶段有效性，建议使用RDB持久化方案

数据可以良好的做到阶段内无丢失（该阶段是开发者或运维人员手工维护的），且恢复速度较快，阶段点数据恢复通常采用RDB方案

注意：利用RDB实现紧凑的数据持久化会使Redis降的很低，慎重。

1. 综合比对

* RDB与AOF的选择实际上是在做一种权衡，每种都有利有弊
* 如不能承受数分钟以内的数据丢失，对业务数据非常敏感，选用AOF
* 如能承受数分钟以内的数据丢失，且追求大数据集的恢复速度，选用RDB
* 灾难恢复选用RDB
* 双保险策略，同时开启 RDB 和 AOF，重启后，Redis优先使用 AOF 来恢复数据，降低丢失数据的量

# 持久化应用场景

## 简述

不是所有的redis数据都需要持久化，因为redis作为缓存数据库，主要的目的还是提高查询速度，如果没有缓存，其实大部分业务都是能做的。其次redis持久化的数据是否丢失以及保存策略的选择是需要平衡的，否则会影响效率，在有些时候redis持久化不能保证数据完全不丢失，因此在有些场景是不需要开启持久化的。

如果部分数据需要持久化，部分数据不需要持久化。那么应该将数据分开存储。即分别存储在不同的redis中。

## 场景举例

Tips 1：redis用于控制数据库表主键id，为数据库表主键提供生成策略，保障数据库表的主键唯一性（不需要，因为数据丢失可能主键冲突，应该从数据库加载值redis中）

Tips 3：redis应用于各种结构型和非结构型高热度数据访问加速（不需要，因为热点数据实时变化的）

Tips 4：redis 应用于购物车数据存储设计（不需要，购物车数据应该存储在数据中）

Tips 5：redis 应用于抢购，限购类、限量发放优惠卷、激活码等业务的数据存储设计

Tips 6：redis 应用于具有操作先后顺序的数据控制

Tips 7：redis 应用于最新消息展示

Tips 9：redis 应用于同类信息的关联搜索，二度关联搜索，深度关联搜索（不需要，层级太深，应该在数据库加载）

Tips 12：redis 应用于基于黑名单与白名单设定的服务控制（如果是限时性质的黑名单，需要持久化，否则不需要）

Tips 13：redis 应用于计数器组合排序功能对应的排名

Tips 15：redis 应用于即时任务/消息队列执行管理（可以选在消息中间件）

Tips 16：redis 应用于按次结算的服务控制（建议不使用，直接在库中加载）

# Redis事物

## redis事物简介

* 释义

redis事务就是一个命令执行的队列，将一系列预定义命令包装成一个整体（一个队列）。当执行时，一次性按照添加顺序依次执行，中间不会被打断或者干扰。

一个队列中，一次性、顺序性、排他性的执行一系列命令。

## 事务基本操作

1. 开启事务

设定事务的开启位置，此指令执行后，后续的所有指令均加入到事务中

|  |
| --- |
| multi |

1. 执行事务

设定事务的结束位置，同时执行事务。与multi成对出现，成对使用

注意：加入事务的命令暂时进入到任务队列中，并没有立即执行，只有执行exec命令才开始执行

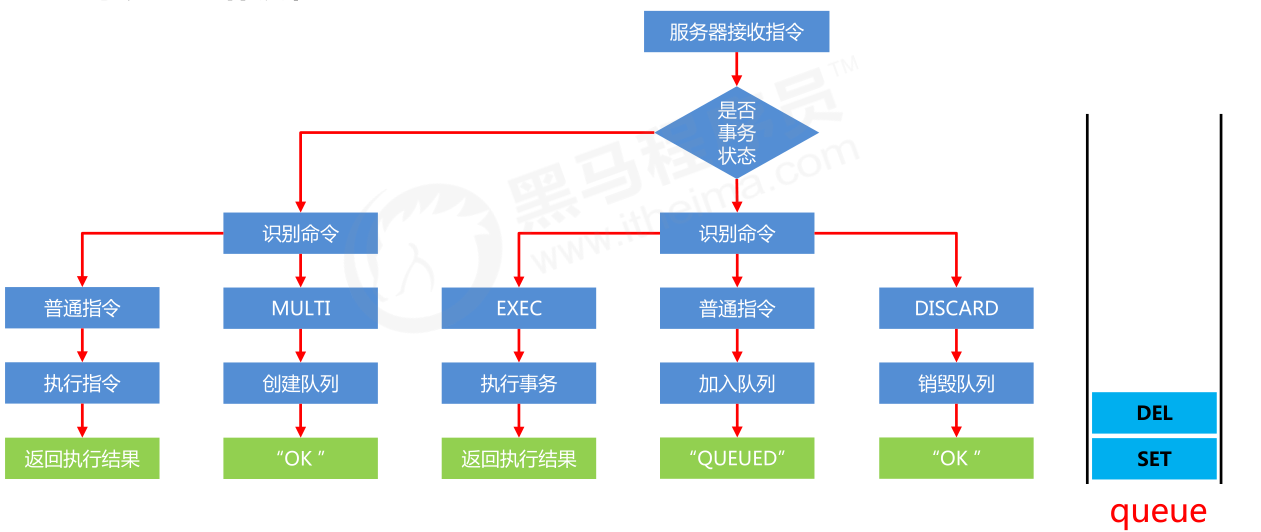
|  |
| --- |
| exec |

1. 取消事务

终止当前事务的定义，发生在multi之后，exec之前

|  |
| --- |
| discard |

## 事物工作流程



## 注意事项（由于redis事物回顾有一定局限性，所以在生成环境很少使用）

1. 语法错误：指命令书写格式有误

如果定义的事务中所包含的命令存在语法错误，整体事务中所有命令均不会执行。包括那些语法正确的命令。

1. 运行错误：指命令格式正确，但是无法正确的执行。例如对list进行incr操作

能够正确运行的命令会执行，运行错误的命令不会被执行

1. 注意：

已经执行完毕的命令对应的数据不会自动回滚，需要程序员自己在代码中实现回滚。

1. 手动进行事务回滚

* 记录操作过程中被影响的数据之前的状态：单数据：string，多数据：hash、list、set、zset
* 设置指令恢复所有的被修改的项：单数据：直接set（注意周边属性，例如时效）。多数据：修改对应值或整体克隆复制

# 事物-锁

## 业务场景

天猫双11热卖过程中，对已经售罄的货物追加补货，4个业务员都有权限进行补货。补货的操作可能是一系列的操作，牵扯到多个连续操作，如何保障不会重复操作？

## 业务分析

1. 多个客户端有可能同时操作同一组数据，并且该数据一旦被操作修改后，将不适用于继续操作。
2. 在操作之前锁定要操作的数据，一旦发生变化，终止当前操作

## 解决方案-锁（基本操作）

* 对 key 添加监视锁，在执行exec前如果key发生了变化，终止事务执行

|  |
| --- |
| watch key1 [key2……] |

* 取消对所有 key 的监视

|  |
| --- |
| unwatch |

* Tips 18

redis 应用基于状态控制的批量任务执行

## 注意事项

1. 监听watch是针对事物的;
2. 监听watch是一次的，即开启一次监听后，无论事物成功或者失败，此次监听都失效。

# redis-分布式锁

## 业务场景

双十一商品抢购，防止超卖问题、红包雨等应用场景

## 分布式锁应考虑如下问题

1. 互斥性。在任意时刻，只有一个客户端能持有锁。
2. 不会发生死锁。即要保证客户端主动解锁，也要保证即使有一个客户端在持有锁的期间崩溃而没有主动解锁，也能保证后续其他客户端能加锁
3. 具有容错性。只要大部分的Redis节点正常运行，客户端就可以加锁和解锁。
4. 防止锁误删。即谁加锁，谁解锁。
5. 重入锁。在同一线程内，外层函数获得锁之后，内层递归函数仍然可以获取到该锁，换一种说法：同一个线程再次进入同步代码时，可以使用自己已获取到的锁。可重入锁可以避免因同一线程中多次获取锁而导致死锁发生

## 解决方案

1. 使用 setnx 设置一个公共锁解决互斥性问题。
2. 通过del操作，以及设置有效期来解决死锁问题。
3. 通过redis高可用，解决容错性问题。
4. 使用value存储线程ID,解锁时需要验证线程ID是否相同，解决防止锁误删问题
5. 使用redis的hash存储线程的重入次数，解决重入锁问题

## 实现代码

* 获取锁的脚本(注释删掉,不然运行报错)

|  |
| --- |
| local key = KEYS[1]; -- 第1个参数,锁的key  local threadId = ARGV[1]; -- 第2个参数,线程唯一标识  local releaseTime = ARGV[2]; -- 第3个参数,锁的自动释放时间  if(redis.call('exists', key) == 0) then -- 判断锁是否已存在  redis.call('hset', key, threadId, '1'); -- 不存在, 则获取锁  redis.call('expire', key, releaseTime); -- 设置有效期  return 1; -- 返回结果  end;  if(redis.call('hexists', key, threadId) == 1) then -- 锁已经存在，判断threadId是否是自己  redis.call('hincrby', key, threadId, '1'); -- 如果是自己，则重入次数+1  redis.call('expire', key, releaseTime); -- 设置有效期  return 1; -- 返回结果  end;  return 0; -- 代码走到这里,说明获取锁的不是自己，获取锁失败 |

* 释放锁的脚本(注释删掉,不然运行报错)

|  |
| --- |
| local key = KEYS[1]; -- 第1个参数,锁的key  local threadId = ARGV[1]; -- 第2个参数,线程唯一标识  if (redis.call('HEXISTS', key, threadId) == 0) then -- 判断当前锁是否还是被自己持有  return nil; -- 如果已经不是自己，则直接返回  end;  local count = redis.call('HINCRBY', key, threadId, -1); -- 是自己的锁，则重入次数-1  if (count == 0) then -- 判断是否重入次数是否已经为0  redis.call('DEL', key); -- 等于0说明可以释放锁，直接删除  return nil;  end; |

* java代码(jedis工具类、支持重入、不支持重入)



* 测试代码



## 锁释放时间设定

由于操作通常都是微秒或毫秒级，因此该锁定时间不宜设置过大。具体时间需要业务测试后确认。

* 例如：持有锁的操作最长执行时间127ms，最短执行时间7ms。
* 测试百万次最长执行时间对应命令的最大耗时，测试百万次网络延迟平均耗时
* 锁时间设定推荐：最大耗时\*120%+平均网络延迟\*110%
* 如果业务最大耗时<<网络平均延迟，通常为2个数量级，取其中单个耗时较长即可

# 删除策略

## 过期数据

1. 查看数据的状态

Redis是一种内存级数据库，所有数据均存放在内存中，内存中的数据可以通过TTL指令获取其状态。

XX ：具有时效性的数据；

-1 ：永久有效的数据

-2 ：已经过期的数据 或 被删除的数据 或 未定义的数据

1. 过期数据定义

在redis中，会单独分配内存空间记录数据的过期时间(expire)

数据结构：key(数据的内存地址值)：value(数据的过期时间)。其中过期数据就是指这部分空间中已经到期的数据，并将其删除，目的是释放内存空间。

1. 过期数据清理的会产生的问题

redis无论是set数据，还是获取数据，或者是清除过期数据，都是需要发送指令给cpu,cpu会处理相应的指令，但是当cpu计算任务过多时，处理效率会变慢，这样会导致主要任务（set数据、get数据） 处理变慢，影响整体业务的响应效率。因此我们理想的状态是，在cpu空闲时，处理过期数据清理的任务，这样就能不影响整体主业务的处理，但是又会带来新的问题，就是内存可能占用过大。**因此我们在删除过期数据时，应在内存占用与CPU占用之间寻找一种平衡，即平衡时间与空间，顾此失彼都会造成整体redis性能的下降，甚至引发服务器宕机或内存泄露。**

## 数据删除策略

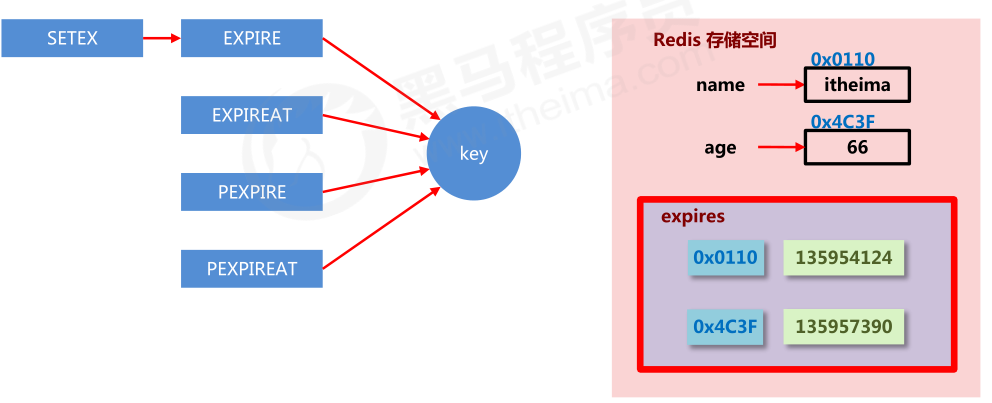
1. 目的

在内存占用与CPU占用之间寻找一种平衡，顾此失彼都会造成整体redis性能的下降，甚至引发服务器宕机或内存泄露

1. 策略

定时删除、惰性删除、定期删除

1. 时效性数据的存储结构



## 定时删除

1. 定义

创建一个定时器，当key设置有过期时间，且过期时间到达时，由定时器任务立即执行对键的删除操作。

1. 优点

节约内存，到时就删除，快速释放掉不必要的内存占用。

1. 缺点

CPU压力很大，无论CPU此时负载量多高，均占用CPU，会影响redis服务器响应时间和指令吞吐量

1. 总结

用处理器性能换取存储空间 （拿时间换空间）

## 惰性删除

1. 定义

数据到达过期时间，不做处理。等下次访问该数据时，如果未过期，则返回数据；如果发现已过期，则删除返回不存在。如果使用此方案，在redis内部执行get操作时都会调用expireIfNeeded()操作

1. 优点

优点：节约CPU性能，发现必须删除的时候才删除

1. 缺点

内存压力很大，出现长期占用内存的数据

1. 总结

用存储空间换气处理器性能（拿空间换时间）

## 定期删除

1. 定义

周期性轮询redis库中的时效性数据，采用随机抽取的策略，利用过期数据占比的方式控制删除频度。

1. 优点

* CPU性能占用设置有峰值，检测频度可自定义设置
* 内存压力不是很大，长期占用内存的冷数据会被持续清理

1. 总结

周期性抽查存储空间（随机抽查，重点抽查）

1. 定期删除过程

* Redis启动服务器初始化时，读取配置server.hz的值，默认为10
* 每秒钟执行server.hz次serverCron() –> databasesCron() -> activeExpireCycle()

其中databasesCron是轮询expires(0-15)

activeExpireCycle是随机抽取expires(\*)内部的数据的进行过期数据处理的

* 对某个expires[\*]检测时，随机挑选W个key检测，逻辑如下：

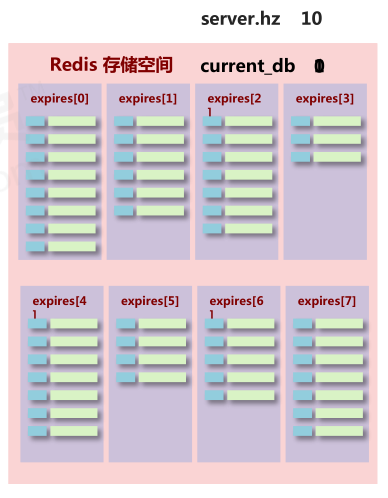
如果key超时，删除key

如果一轮中删除的key的数量>W\*25%，循环该过程

如果一轮中删除的key的数量≤W\*25%，检查下一个expires[\*]，0-15循环

W取值=ACTIVE\_EXPIRE\_CYCLE\_LOOKUPS\_PER\_LOOP属性值

* 参数current\_db用于记录activeExpireCycle() 进入哪个expires[\*] 执行
* 如果activeExpireCycle()执行时间到期，下次从current\_db继续向下执行



## 删除策略比对

|  |  |  |  |
| --- | --- | --- | --- |
| **策略** | **内存占用** | **cpu占用** | **总结** |
| 定时删除 | 节约内存，无占用 | 不分时段占用CPU资源，频度高 | 拿时间换空间 |
| 惰性删除 | 内存占用严重 | 延时执行，CPU利用率高 | 拿空间换时间 |
| 定期删除 | 内存定期随机清理 | 每秒花费固定的CPU资源维护内存 | 随机抽查，重点抽查 |

## 逐出算法

1. **定义**

Redis使用内存存储数据，在执行每一个命令前，会调用freeMemoryIfNeeded()检测内存是否充足。如果内存不满足新加入数据的最低存储要求，redis要临时删除一些数据为当前指令清理存储空间。清理数据的策略称为**逐出算法**。

1. **注意**

逐出数据的过程不是100%能够清理出足够的可使用的内存空间，如果不成功则反复执行。当对所有数据尝试完毕后，如果不能达到内存清理的要求，将出现错误信息。

|  |
| --- |
| (error) OOM command not allowed when used memory >'maxmemory' |

1. **影响数据逐出的相关配置**

* 最大可使用内存

占用物理内存的比例，默认值为0，表示不限制。生产环境中根据需求设定，通常设置在50%以上。

|  |
| --- |
| maxmemory |

* 每次选取待删除数据的个数

选取数据时并不会全库扫描，导致严重的性能消耗，降低读写性能。因此采用随机获取数据的方式作为待检测删除数据

|  |
| --- |
| maxmemory-samples |

* 删除策略

达到最大内存后的，对被挑选出来的数据进行删除的策略

|  |
| --- |
| maxmemory-policy |

1. **8中删除策略**

* 检测易失数据（可能会过期的数据集server.db[i].expires ）

1. volatile-lru：挑选最近最少使用的数据淘汰**（推荐）**
2. volatile-lfu：挑选最近使用次数最少的数据淘汰
3. volatile-ttl：挑选将要过期的数据淘汰
4. volatile-random：任意选择数据淘汰

* 检测全库数据（所有数据集server.db[i].dict ）

1. allkeys-lru：挑选最近最少使用的数据淘汰
2. allkeys-lfu：挑选最近使用次数最少的数据淘汰
3. allkeys-random：任意选择数据淘汰

* 放弃数据驱逐

no-enviction（驱逐）：禁止驱逐数据（redis4.0中默认策略），会引发错误OOM（Out Of Memory）

1. **数据逐出策略配置依据**

使用INFO命令输出监控信息，查询缓存 hit 和 miss 的次数，根据业务需求调优Redis配置

# Redis核心配置

## 服务器端设定

1. 设置服务器以守护进程的方式运行

|  |
| --- |
| daemonize yes|no |

1. 绑定主机地址

|  |
| --- |
| bind 127.0.0.1 |

1. 设置服务器端口号

|  |
| --- |
| port 6379 |

1. 设置数据库数量

|  |
| --- |
| databases 16 |

## 日志配置

1. 设置服务器以指定日志记录级别

|  |
| --- |
| loglevel debug|verbose|notice|warning |

1. 日志记录文件名

|  |
| --- |
| logfile 端口号.log |

1. 注意：日志级别开发期设置为verbose即可，生产环境中配置为notice，简化日志输出量，降低写日志IO的频度

## 客户端配置

1. 设置同一时间最大客户端连接数，默认无限制。当客户端连接到达上限，Redis会关闭新的连接

|  |
| --- |
| maxclients 0 |

1. 客户端闲置等待最大时长，达到最大值后关闭连接。如需关闭该功能，设置为 0

|  |
| --- |
| timeout 300 |

## 多服务器快捷配置

1. 导入并加载指定配置文件信息，用于快速创建redis公共配置较多的redis实例配置文件，便于维护

|  |
| --- |
| include /path/server-端口号.conf |

# 高级数据类型

高级数据类型的产生，是建立在已有的数据类型上，但是是为了解决某种特定问题的，例如Bitmaps是为了解决信息状态统计的。HyperLogLog做基数统计。GEO坐标之间计算的

## Bitmaps

1. **出现背景**

有些业务场景，是统计业务的状态（只能统计状态，即是否等待），例如：

* 统计每天某一部电影是否被点播
* 统计每天有多少部电影被点播
* 统计每周/月/年有多少部电影被点播
* 统计年度哪部电影没有被点播
* 年度浏览量最低（查询状态，用程序统计）
* 月度浏览量最低（查询状态，用程序统计）

1. **基础操作**

* 获取指定key对应偏移量上的bit值

getbit key offset

* 设置指定key对应偏移量上的bit值，value只能是1或0

setbit key offset value

1. **扩展操作**

* 对指定key按位进行交、并、非、异或操作，将结果保存到destKey中

bitop op destKey key1 [key2...]

and：交

or：并

not：非

xor：异或

* 统计指定key中1的数量

bitcount key [start end]

1. **例如：记录《非诚勿扰》的id:5 （offset:4）的20200909是否被浏览**

setbit 20200909 4 1

setbit 20200909 5 1

1. **Tips 21：**

redis 应用于信息状态统计

## HyperLogLog

1. 出现背景

需要统计大量数量的不重复数据的数量，原始方案：set, 存储每个用户的id（字符串）。改进方案：Bitmaps，存储每个用户状态（bit）。但是这2中方案，如果数据量太大时，存储的空间也会很大，因此出现全新的方案：Hyperloglog

1. 基础操作

* 添加数据

pfadd key element [element ...]

* 统计数据

pfcount key [key ...]

* 合并数据

pfmerge destkey sourcekey [sourcekey...]

1. 相关说明

* 用于进行基数统计，不是集合，不保存数据，只记录数量而不是具体数据
* 核心是基数估算算法，最终数值存在一定误差
* 误差范围：基数估计的结果是一个带有 0.81% 标准错误的近似值
* 耗空间极小，每个hyperloglog key占用了12K的内存用于标记基数
* pfadd命令不是一次性分配12K内存使用，会随着基数的增加内存逐渐增大
* Pfmerge命令合并后占用的存储空间为12K，无论合并之前数据量多少

## GEO

1. **出现背景**

火热的生活服务类软件，例如：微信 / 陌陌、美团 / 饿了么、携程 / 马蜂窝、高德 / 百度等等，需要展现附近有多少人，每个人离你100米以内等等一些统计信息

1. **基本操作（需要将需要计算的坐标放入一组中，才能计算）**

* 添加坐标点

geoadd key longitude latitude member [longitude latitude member ...]

* 获取坐标点（key相当于组名，member,在组中的成员名）

geopos key member [member ...]

* 计算坐标点距离（计算组中A点，B点的距离，可以设置显示单位）

geodist key member1 member2 [unit]

* 输入坐标点、距离半径。计算符合条件的组中成员

georadius key longitude latitude radius m|km|ft|mi [withcoord] [withdist] [withhash] [count count]

* 输入组中成员名、距离半径。计算符合条件的组中成员

georadiusbymember key member radius m|km|ft|mi [withcoord] [withdist] [withhash] [count count]

* 计算组中成员的hash值

geohash key member [member ...]

1. **Tips 23：**

redis 应用于地理位置计算

# 主从复制

## 主从复制简介

1. 高可用定义

（全年时间 – 服务全年的不可用的时间）/ 全年时间 \* 100 = 99.99%

4个9,是我们最求高可用的目标，即高可用，就是今最大努力减少服务不可用的时间。

1. 单击Redis的风险与问题

* 问题1：机器故障

例如硬盘故障、系统崩溃等等，造成数据丢失，可能对业务造成灾难性打击。最终有可能放弃使用Redis

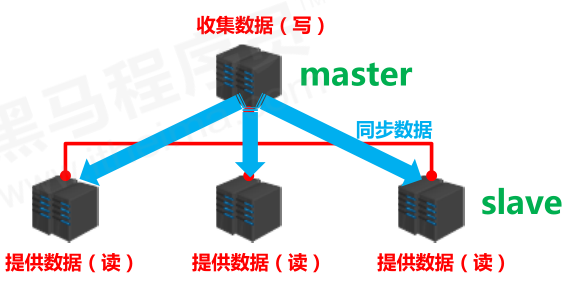
* 问题2：容量瓶颈

内存不足，物理机从16G升级到64G，从64G升级到128G，但是服务器不能能无限升级。这样会导致硬件成本过高。最终有可能放弃使用redis

1. 解决单机Redis方案

为了避免单点Redis服务器故障，准备多台服务器，互相连通。将数据复制多个副本保存在不同的服务器上，连接在一起，并保证数据是同步的。即使有其中一台服务器宕机，其他服务器依然可以继续提供服务，实现Redis的高可用，同时实现数据冗余备份。

1. 主从复制架构



1. 主从复制名词解释

* 提供数据方：master、主服务器、主节点、主库，客户端为主客户端
* 接收数据方：slave、从服务器、从节点、从库，客户端为从客户端
* 需要解决的问题：数据同步
* 核心工作：master的数据复制到slave中

1. 主从复制基本要求

* 主从复制即将master中的数据即时、有效的复制到slave中
* 特征：一个master可以拥有多个slave，一个slave只对应一个master
* master：写数据、执行写操作时，将出现变化的数据自动同步到slave,读数据（可忽略）
* slave：读数据，写数据（禁止）

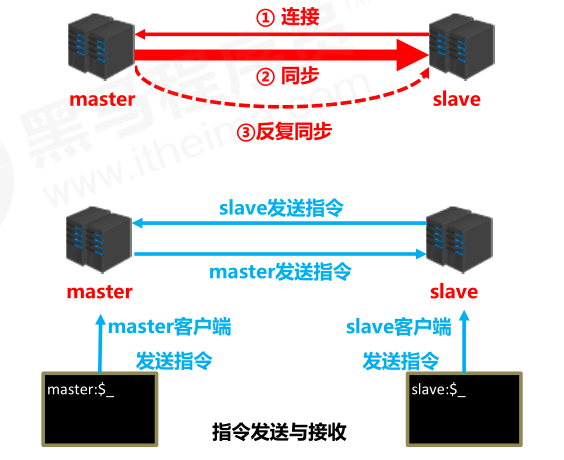
1. 主从复制的作用

* 读写分离，分担写的压力：master写、slave读，提高服务器的读写负载能力
* 基于主从结构，配合读写分离，由slave分担master负载，并根据需求的变化，改变slave的数量，通过多个从节点分担数据读取负载，大大提高Redis服务器并发量与数据吞吐量
* 故障恢复：当master出现问题时，由slave提供服务，实现快速的故障恢复
* 数据冗余：实现数据热备份，是持久化之外的一种数据冗余方式
* 高可用基石：基于主从复制，构建哨兵模式与集群，实现Redis的高可用方案

## 主从复制工作流程（3个阶段）

1. 主从复制过程大体可以分为3个阶段

* 第一阶段：建立连接阶段（即准备阶段）
* 第二阶段：数据同步阶段
* 第三阶段：命令传播阶段



## 主从复制阶段一：建立连接阶段

1. 此阶段目的

建立slave到master的连接，使master能够识别slave，并保存slave端口号

1. 建立连接阶段工作流程

步骤1：slave设置master的地址和端口，得到master成功响应后，保存master信息

步骤2：slave建立socket连接

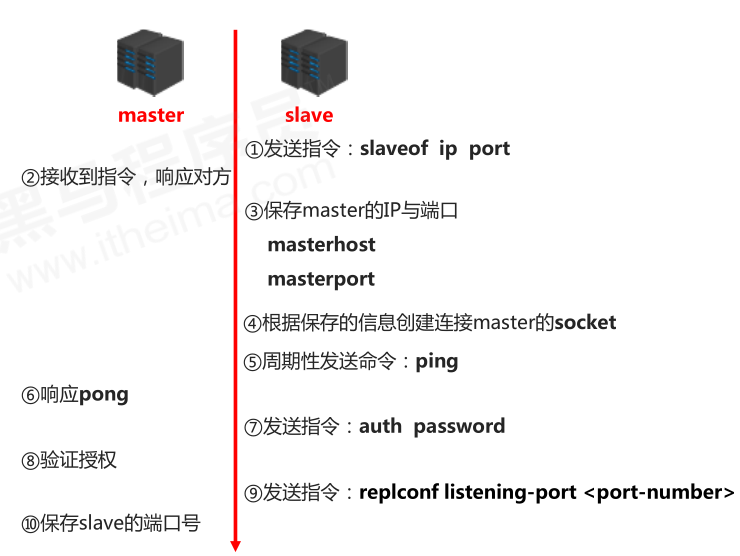
步骤3：slave发送ping命令（定时器任务）,master成功响应pong

步骤4：slave发送身份验证

步骤5：slave发送slave端口信息，

步骤6：master保存slave的端口号，

至此，主从连接成功！



1. 主从连接成功后的状态信息

slave:保存master的地址与端口。

master：保存slave的端口

总体：slave与master之间创建了连接的socket

1. 主从连接（slave连接master）命令

* 方式一：客户端发送命令（不常用）

slaveof <masterip> <masterport>

* 方式二：启动服务器参数（不常用）

redis-server -slaveof <masterip> <masterport>

* **方式三：服务器配置（常用）**

slaveof <masterip> <masterport>

如果master有权限，则设置

masterauth <password>

* 查看slave系统信息（连接成功）显示如下：info

master\_link\_down\_since\_seconds

masterhost

masterport

* 查看master系统信息（连接成功）显示如下：info

slave\_listening\_port(多个)

* 主从断开连接（从客户端发送命令）

slaveof no one

说明：slave断开连接后，不会删除已有数据，只是不再接受master发送的数据

1. 授权访问

* 客户端发送命令设置密码

config set requirepass <password>

config get requirepass

* 配置文件设置密码

requirepass <password>

* 客户端发送权限验证

auth <password>

* slave配置文件设置主服务器密码

masterauth <password>

## 主从复制阶段二：数据同步（关键看说明）

1. 全量复制

用于初次复制或其它无法进行部分复制的情况，将主节点中的所有数据都发送给从节点。当数据量过大的时候，会造成很大的网络开销。

1. 部分复制

用于处理在主从复制中因网络闪退等原因造成数据丢失场景，当从节点再次连上主节点，如果条件允许，主节点会补发丢失数据给从节点，因为补发的数据远远小于全量数据，可以有效避免全量复制的过高开销。但需要注意，如果网络中断时间过长，造成主节点没有能够完整地保存中断期间执行的写命令，则无法进行部分复制，仍使用全量复制 。

1. 复制偏移量

* 参与复制的主从节点都会维护自身复制偏移量，主节点在处理完写入命令操作后，会把命令的字节长度做累加记录，统计信息在info replication中的master\_repl\_offset指标中。
* 从节点每秒钟上报自身的复制偏移量给主节点，因此主节点也会保存从节点的复制偏移量slave0:ip=192.168.1.3,port=6379,

state=online,offset=116424,lag=0

* 从节点在接收到主节点发送的命令后，也会累加记录自身的偏移量。统计信息在info replication中的slave\_repl\_offset中

1. 复制积压缓冲区

* 复制积压缓冲区是保存在主节点上的一个固定长度的队列，默认大小为1MB，**当主节点有连接的从节点时被创建**，这时主节点响应写命令时，不但会把命令发给从节点，还会写入复制积压缓冲区。
* 在命令传播阶段，主节点除了将写命令发送给从节点，还会发送一份给复制积压缓冲区，作为写命令的备份；除了存储写命令，复制积压缓冲区中还存储了其中 的每个字节对应的复制偏移量(offset) 。由于复制积压缓冲区定长且先进先出，所以它保存的是主节点最近执行的写命令；时间较早的写命令会被挤出缓冲区。

1. 主节点运行ID

每个redis节点启动后都会动态分配一个40位的十六进制字符串为运行ID。运行ID的主要作用是来唯一识别redis节点，比如从节点保存主节点的运行ID识别自已正在复制是哪个主节点。如果只使用ip+port的方式识别主节点，那么主节点重启变更了整体数据集（如替换RDB/AOF文件），从节点再基于偏移量复制数据将是不安全的，因此当运行ID变化后从节点将做全量复制。可以在info server命令查看当前节点的运行ID。

需要注意的是redis关闭再启动，运行的id会随之变化。

1. Psync2命令（数据同步命令）

在2.8版本之前是sync命令，2.8-4.0版本是Psync命令，4.0以后优化为Psync2命令

流程说明：

1. 从节点(slave)发送psync命令给主节点，参数runid是当前从节点保存的主节点运行id，如果没有则默认值为 ？, 参数offset是当前从节点保存的复制偏移量，如果是第一次参与复制则默认值为-1。
2. 主节点根据pysnc参数和自身数据情况决定响应结果，从节点工具主节点返回结果执行全量复制或者部分复制：

* 如果回复+FULLRESYNC {runid} {offset}，那么从节点将触发全量复制流程。
* 如果回复+CONTINUE，从节点将触发部分复制流程。
* 如果回复-ERR，说明主节点版本低于Redis2.8

1. **数据同步工作流程**

步骤1：请求同步数据

步骤2：创建RDB同步数据

步骤3：恢复RDB同步数据

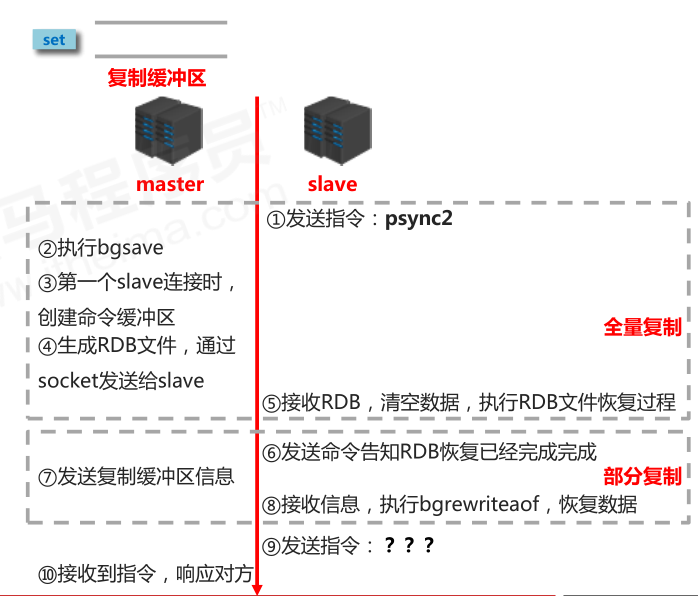
步骤4：请求部分同步数据

步骤5：恢复部分同步数据

至此，数据同步工作完成！

状态：slave：具有master端全部数据，包含RDB过程接收的数据

master：保存slave当前数据同步的位置



1. 数据同步阶段master说明

* 如果master数据量巨大，数据同步阶段应避开流量高峰期，避免造成master阻塞，影响业务正常执行
* 复制缓冲区大小设定不合理，会导致数据溢出。如进行全量复制周期太长，进行部分复制时发现数据已经存在丢失的情况，必须进行第二次全量复制，致使slave陷入死循环状态。

repl-backlog-size 1mb

* master单机内存占用主机内存的比例不应过大，建议使用50%-70%的内存，留下30%-50%的内存用于执行bgsave命令和创建复制缓冲区
* 网络延迟问题

主从节点一般部署在不同机器上，复制时的网络延迟就成为需要考虑的问题，redis为我们提供了repl-disable-tcp-nodelay参数用于控制是否关闭 tcp nodelay，默认是关闭的。

当关闭时，主节点产生的命令数据无论大小都会及时地发送给从节点，这样主从之间延迟将会变小，但增加了网络宽带的消耗。适用于主从之间的网络环境较好的场景。

当开启时，主节点会合并较小的TCP数据包从而节省宽带。默认发送时间间隔取决于Linux的内核，一般默认为40ms。这种配置节省了宽带但增大主从之间的延迟。适用于主从网络环境复杂或宽带紧张的场景。

1. 数据同步阶段slave说明

* 为避免slave进行全量复制、部分复制时服务器响应阻塞或数据不同步，建议关闭此期间的对外服务（慎用）

|  |
| --- |
| slave-serve-stale-data yes|no |

* 数据同步阶段，master发送给slave信息可以理解master是slave的一个客户端，主动向slave发送命令
* 多个slave同时对master请求数据同步，master发送的RDB文件增多，会对带宽造成巨大冲击，如果master带宽不足，因此数据同步需要根据业务需求，适量错峰
* slave过多时，建议调整拓扑结构，由一主多从结构变为树状结构，中间的节点既是master，也是slave。注意使用树状结构时，由于层级深度，导致深度越高的slave与最顶层master间数据同步延迟较大，数据一致性变差，应谨慎选择

## 命令传播阶段相关概念（3个核心）

1. 命令传播阶段可能出现的问题

命令传播阶段是在数据同步阶段完成之后进行的，即同步master与salve新增数据。因为网络不受Redis控制，因此会产生如下问题：

* 网络闪断闪连：这种情况暂时忽略
* 短时间网络中断：部分复制（复制积压缓冲区与偏移量实现的）
* 长时间网络中断：全量复制

1. 部分复制的三个核心要素

* 服务器的运行id（run id）
* 主服务器的复制积压缓冲区
* 主从服务器的复制偏移量

1. **服务器的运行ID（run id）**

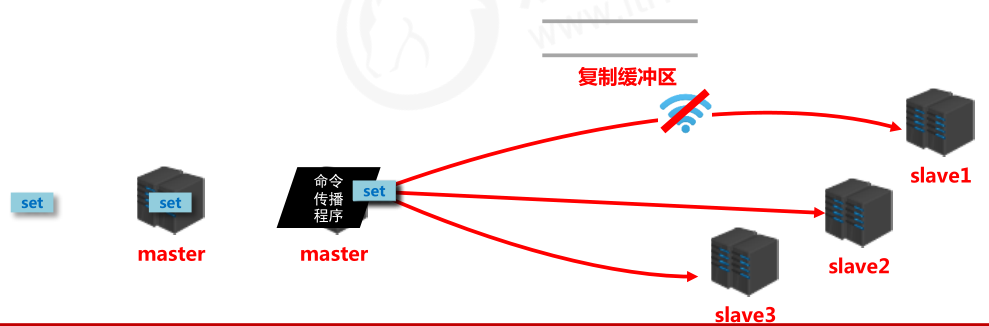
* 概念：服务器运行ID是每一台服务器每次运行的身份识别码，一台服务器多次运行可以生成多个运行id
* 组成：运行id由40位字符组成，是一个随机的十六进制字符,

例如：fdc9ff13b9bbaab28db42b3d50f852bb5e3fcdce

* 作用：运行id被用于在服务器间进行传输，识别身份。如果想两次操作均对同一台服务器进行，必须每次操作携带对应的运行id，用于对方识别
* 运行id在每台服务器启动时自动生成的，master在首次连接slave时，会将自己的运行ID发送给slave，slave保存此ID，通过info Server命令，可以查看节点的runid

1. 复制缓冲区

* 概念：复制缓冲区，又名复制积压缓冲区，是一个先进先出（FIFO）的队列，用于存储服务器执行过的命令，每次传播命令，master都会将传播的命令记录下来，并存储在复制缓冲区。



* 组成：偏移量、字节值
* 工作原理：1.通过offset区分不同的slave当前数据传播的差异。2. master记录已发送的信息对应的offset。3. slave记录已接收的信息对应的offset
* 与AOF存储一样：

|  |
| --- |
| $3  set  $4  name  $7  itheima  $3 \r \n  set \r \n  $4 \r \n  name \r \n  $7 \r \n  itheima \r \n |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 偏移量 | 9041 | 9042 | 9043 | 9044 | 9045 | 9046 | 9047 | 9048 | 9049 |
| 字节值 | ... | $ | 3 | \r | \n | s | e | t | ... |

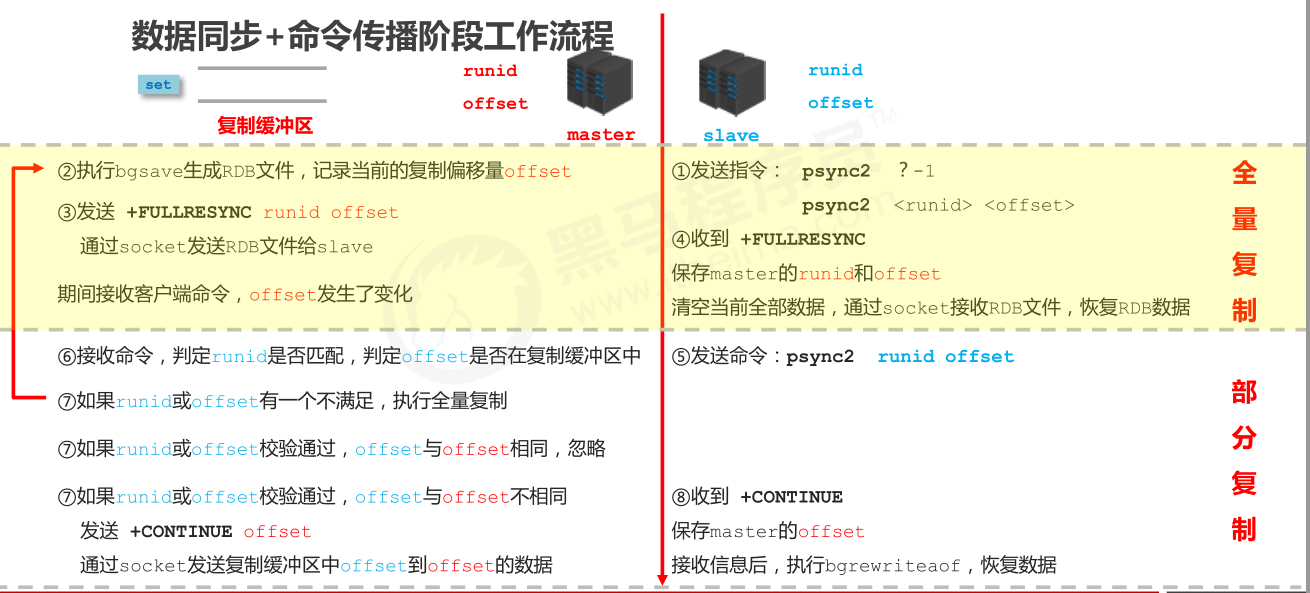
**复制缓冲区/复制积压缓冲区**

* 复制缓冲区默认数据存储空间大小是1M，由于存储空间大小是固定的，当入队元素的数量大于队列长度时，最先入队的元素会被弹出，而新元素会被放入队列。
* 由来：每台服务器启动时，如果开启有AOF或被连接成为master节点，即创建复制缓冲区
* 作用：用于保存master收到的所有指令（仅影响数据变更的指令，例如set，select）
* 数据来源：当master接收到主客户端的指令时，除了将指令执行，会将该指令存储到缓冲区中

1. 主从服务器复制偏移量（offset）

* 概念：一个数字，描述复制缓冲区中的指令字节位置
* master复制偏移量：记录发送给所有slave的指令字节对应的位置（多个）
* slave复制偏移量：记录slave接收master发送过来的指令字节对应的位置（一个）
* master端：发送一次记录一次
* slave端：接收一次记录一次
* 作用：同步信息，比对master与slave的差异，当slave断线后，恢复数据使用

## 数据同步+命令传播阶段工作流程（二、三阶段详细）



## 心跳机制

* 进入命令传播阶段候，master与slave间需要进行信息交换，使用心跳机制进行维护，实现双方连接保持在线
* master心跳：

指令：PING

周期：由repl-ping-slave-period决定，默认10秒

作用：判断slave是否在线

查询：INFO replication 获取slave最后一次连接时间间隔，lag项维持在0或1视为正常

* slave心跳任务

指令：REPLCONF ACK {offset}

周期：1秒

作用1：汇报slave自己的复制偏移量，获取最新的数据变更指令

作用2：判断master是否在线

* 当slave多数掉线，或延迟过高时，master为保障数据稳定性，将拒绝所有信息同步操作

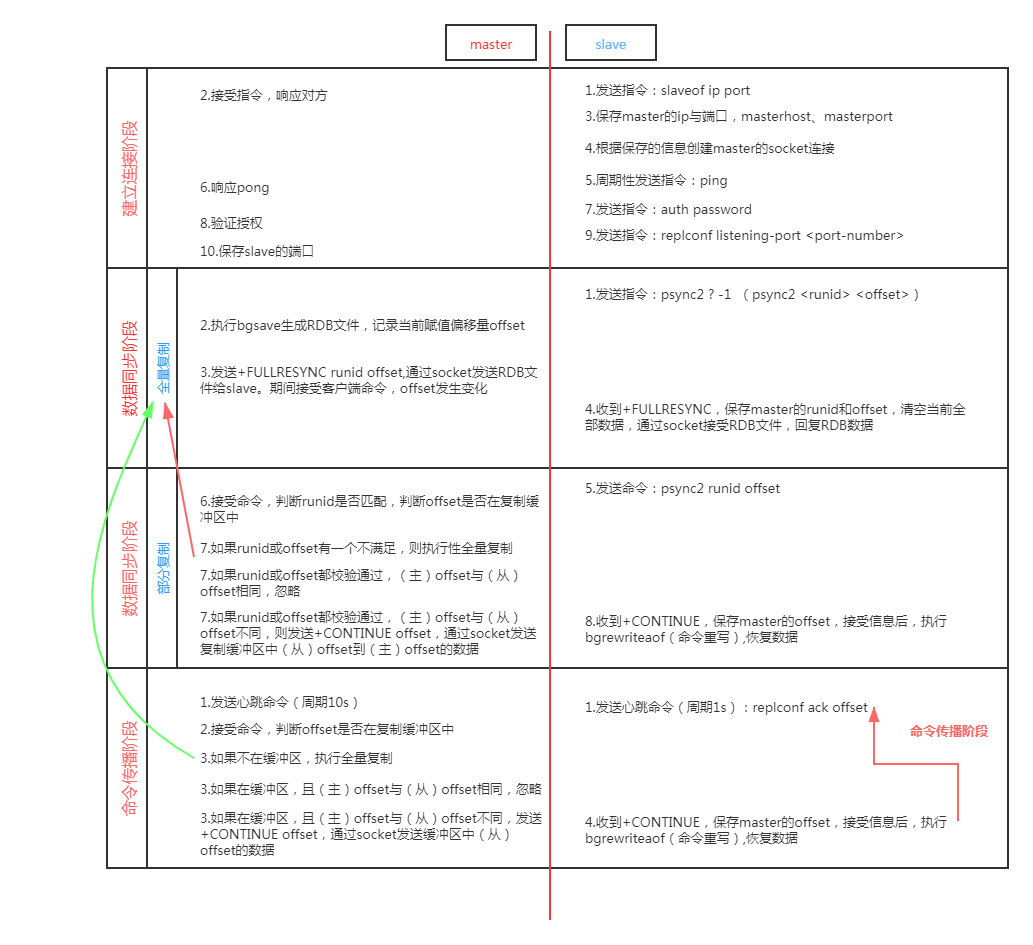
|  |
| --- |
| min-slaves-to-write 2  min-slaves-max-lag 8 |

slave数量少于2个，或者所有slave的延迟都大于等于10秒时，强制关闭master写功能，停止数据同步

slave数量由slave发送REPLCONF ACK命令做确认

slave延迟由slave发送REPLCONF ACK命令做确认

## 主从复制全流程



# 主从复制常见问题

## 频繁的全量复制（1）

**master频繁启动**

* 现象：

伴随着系统的运行，master的数据量会越来越大，一旦master重启，runid将发生变化，会导致全部slave的全量复制操作

* 内部优化调整方案：

1. master内部创建master\_replid变量，使用runid相同的策略生成，长度41位，并发送给所有slave
2. 在master关闭时执行命令 shutdown save，进行RDB持久化,将runid与offset保存到RDB文件中（repl-id、repl-offset），通过redis-check-rdb命令可以查看该信息
3. master重启后加载RDB文件，恢复数据，重启后，将RDB文件中保存的repl-id与repl-offset加载到内存中（master\_repl\_id = repl、master\_repl\_offset = repl-offset）通过info命令可以查看该信息
4. 作用：

本机保存上次runid，重启后恢复该值，使所有slave认为还是之前的master

## 频繁的全量复制（2）

网络环境不佳，出现网络中断，复制缓冲区过小

1. 问题现象：

网络环境不佳，出现网络中断，slave不提供服务

1. 问题原因：

复制缓冲区过小，断网后slave的offset越界，触发全量复制

1. 最终结果

slave反复进行全量复制

1. 解决方案

修改复制缓冲区大小：repl-backlog-size

1. 建议设置如下：

* 测算从master到slave的重连平均时长second
* 获取master平均每秒产生写命令数据总量

write\_size\_per\_second

* 最优复制缓冲区空间 = 2 \* second \* write\_size\_per\_second

## 频繁的网络中断（1）

* 问题现象：

master的CPU占用过高 或 slave频繁断开连接

* 问题原因：

1. slave每1秒发送REPLCONF ACK命令到master
2. 当slave接到了慢查询时（keys \* ，hgetall等），会大量占用CPU性能
3. master每1秒调用复制定时函数replicationCron()，比对slave发现长时间没有进行响应

* 最终结果

master各种资源（输出缓冲区、带宽、连接等）被严重占用

* 解决方案

通过设置合理的超时时间，确认是否释放slave

|  |
| --- |
| repl-timeout |

该参数定义了超时时间的阈值（默认60秒），超过该值，释放slave

## 频繁的网络中断（2）

* 问题现象

slave与master连接断开

* 问题原因

1. master发送ping指令频度较低
2. master设定超时时间较短
3. ping指令在网络中存在丢包

* 解决方案

提高ping指令发送的频度

|  |
| --- |
| repl-ping-slave-period |

超时时间repl-time的时间至少是ping指令频度的5到10倍，否则slave很容易判定超时，导致master舍弃slave

## 数据不一致

* 问题现象

多个slave获取相同数据不同步

* 问题原因

网络信息不同步，数据发送有延迟

* 解决方案

1. 优化主从间的网络环境，通常放置在同一个机房部署，如使用阿里云等云服务器时要注意此现象
2. 监控主从节点延迟（通过offset）判断，如果slave延迟过大，暂时屏蔽程序对该slave的数据访问

|  |
| --- |
| slave-serve-stale-data yes|no |

开启后仅响应info、slaveof等少数命令（慎用，除非对数据一致性要求很高）

# 哨兵模式

## 环境部署说明

* 服务器10.4.120.28（/usr/local/redis/）：

redis的master,配置文件bin/conf/redis-6379.conf

redis的哨兵26379、26380、26381，配置文件在bin/conf下

* 服务器10.4.120.27（/usr/local/redis/）：

redis的slave的redis-6379.conf,redis-6380.conf

* 总体架构为：一主二从三哨兵

## 哨兵简介

* 如果master宕机，手动我们应该怎么处理

1. 关闭master和所有的slave
2. 找一个slave作为master
3. 修改其他slave配置，连接新的master
4. 启动新的master和所有slave
5. 全量复制\*N+部分复制\*N

* 手动处理，会出现以下问题

1. 关闭期间数据服务谁来承接？
2. 找一个master，怎么找？
3. 修改配置后，原始的master回复怎么办？
4. 怎么让外界无感知，即无感知切换master？

* 哨兵

哨兵(sentinel) 是一个分布式系统，用于对主从结构中的每台服务器进行**监控**，当出现故障时通过投票机制**选择**新的master并将所有slave连接到新的master。

* 哨兵的作用

1. 监控：不断的检查master和slave是否正常运行。master存活检测、master与slave运行情况检测
2. 通知（提醒）：当被监控的服务器出现问题时，向其他（哨兵间，客户端）发送通知。
3. 自动故障转移：断开master与slave连接，选取一个slave作为master，将其他slave连接到新的master，并告知客户端新的服务器地址
4. 注意：哨兵也是一台redis服务器，只是不提供数据服务通常哨兵配置数量为单数

## 哨兵配置参数

1. sentinel auth-pass <服务器名称> <password>

范例：sentinel auth-pass mymaster neusoft

说明：连接服务器口令

1. sentinel monitor <自定义服务名称><主机地址><端口><主从服务器总量>

范例：sentinel monitor mymaster 192.168.194.131 6381 1

说明：设置哨兵监听的主服务器信息，最后的参数决定了最终参与选举的服务器数量（-1）

1. sentinel down-after-milliseconds <服务名称><毫秒数（整数）>

范例：sentinel down-after-milliseconds mymaster 3000

说明：指定哨兵在监控Redis服务时，判定服务器挂掉的时间周期，默认30秒（30000），也是主从切换的启动条件之一

1. sentinel parallel-syncs <服务名称><服务器数（整数）>

范例：sentinel parallel-syncs mymaster 1

说明：指定同时进行主从的slave数量，数值越大，要求网络资源越高，要求约小，同步时间约长

1. sentinel failover-timeout <服务名称><毫秒数（整数）>

范例：sentinel failover-timeout mymaster 9000

说明：指定出现故障后，故障切换的最大超时时间，超过该值，认定切换失败，默认3分钟

1. sentinel notification-script <服务名称><脚本路径>

说明：服务器无法正常联通时，设定的执行脚本，通常调试使用。

## 哨兵结构搭建

哨兵就是一台redis服务，只是有单独的配置文件，启动命令不同，不提供数据服务通常哨兵配置数量为单数

1. 拷贝哨兵配置文件

在解压的redis安装文件中/usr/local/download/redis-5.0.3中有提供的默认哨兵配置文件sentinel.conf。执行如下命令，复制到redis的安装目录中。(grep –v "#"，去掉注释，grep –v "^$"，去掉空行)

|  |
| --- |
| cat sentinel.conf | grep –v "#" | grep –v "^$" > /usr/local/redis/bin/conf/sentinel-26379.conf |

1. 修改哨兵配置文件

daemonize yes

logfile "sentinel-26379.log"

dir ./data

sentinel monitor mymaster 127.0.0.1 6379 2

sentinel auth-pass mymaster neusoft

1. 复制替换2份哨兵配置文件

行编辑，s替换命令，g全局替换，‘/匹配表达式/replacement/’表达式

sed 's/26379/26381/g' sentinel-26379.conf > sentinel-26381.conf

1. 搭建主服务（10.4.120.28）

配置文件

|  |
| --- |
| port 6379  daemonize yes  logfile "6379.log"  dir ./data  dbfilename dump-6379.rdb  rdbcompression yes  rdbchecksum yes  save 10 2  appendonly yes  appendfsync always  appendfilename appendonly-6379.aof  bind 0.0.0.0  requirepass neusoft  protected-mode no  maxmemory-policy volatile-lru  masterauth neusoft |

1. 搭建salve服务（10.4.120.27）6379、6380

修改配置文件

|  |
| --- |
| port 6379  daemonize yes  logfile "6379.log"  dir ./data  bind 0.0.0.0  requirepass neusoft  protected-mode no  maxmemory-policy volatile-lru  **slaveof 10.4.120.28 6379**  **masterauth neusoft** |

1. 启动redis服务

先启动master

再启动slave

最后启动哨兵

|  |
| --- |
| ./redis-sentinel ./conf/sentinel-端口号.conf |

1. 验证哨兵是否启动成功
2. 使用redis-cli –p 26379 ，连接哨兵客户端，输入命令info,查看哨兵信息。出现如下信息：（**slaves=2,sentinels=3,2个slave,3个哨兵**）

|  |
| --- |
| # Sentinel  sentinel\_masters:1  sentinel\_tilt:0  sentinel\_running\_scripts:0  sentinel\_scripts\_queue\_length:0  sentinel\_simulate\_failure\_flags:0  **master0:name=mymaster,status=ok,address=127.0.0.1:6379,slaves=2,sentinels=3** |

1. 查看哨兵配置文件，会多出如下信息（红色部分，为其它哨兵的信息，蓝色部分为其它redis的slave的信息）

|  |
| --- |
| port 26379  daemonize yes  #pidfile /var/run/redis-sentinel.pid  logfile "sentinel-26379.log"  dir "/usr/local/redis/bin/data"  **sentinel myid 1945d418e9df17775060e9bd28b575bbe8353b25**  sentinel deny-scripts-reconfig yes  sentinel monitor mymaster 127.0.0.1 6379 2  sentinel auth-pass mymaster neusoft  sentinel config-epoch mymaster 0  sentinel leader-epoch mymaster 0  **# Generated by CONFIG REWRITE**  protected-mode no  **sentinel known-replica mymaster 10.4.120.27 6379**  **sentinel known-replica mymaster 10.4.120.27 6380**  **sentinel known-sentinel mymaster 127.0.0.1 26380 80282f59834d6b852977d8b73955cdd72f99d93d**  **sentinel known-sentinel mymaster 127.0.0.1 26381 a3dbd708047b1c6ee736f377cc00050b309cb7e5**  sentinel current-epoch 0 |

1. 测试主机宕机，哨兵是否能自动切换

* 如果正常，在哨兵中可以发现如下信息：

1. 其中一台哨兵发现 +sdown master mymaster 127.0.0.1 6379 宕机
2. 通知所有哨兵，+odown master mymaster 127.0.0.1 6379 宕机
3. 哨兵选举new-epoch
4. 切换主机 +switch-master mymaster 127.0.0.1 6379 10.4.120.27 6379

* 如果不正常，没有切换成功：redis failover-abort-no-good-slave

1. 检查master、slave配置文件bind地址是否改为 0.0.0.0
2. 检查master、slave配置文件是否配置masterauth neusoft
3. 检查主从服务器的防火墙是否关闭，或者是否开放端口

# 哨兵工作原理

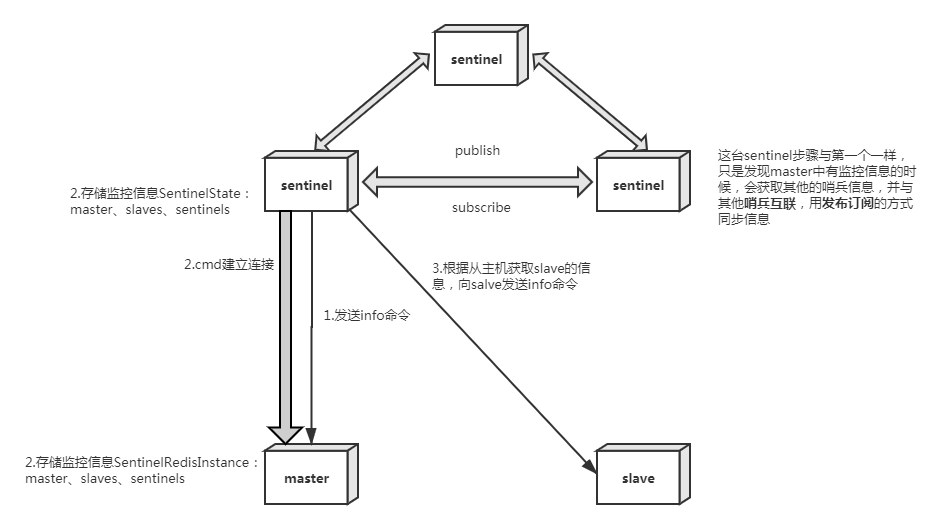
哨兵在进行主从切换工程中经历三个阶段：监控、通知、故障转移。

## 阶段一：监控阶段

1. 同步各个节点的状态信息

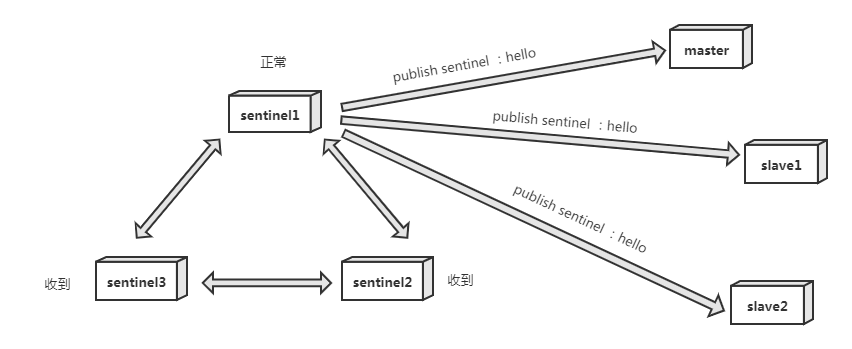
* 获取各个sentinel的状态（是否在线）
* 获取master的状态：master属性（runid、role:master）；获取各个slave的详细信息（因为master存储各个slave的信息ip、port等）
* 根据master中slave的信息，获取所有slave的状态：slave属性（runid、role：slave、master\_host、master\_port、offset等）

1. 监控阶段流程



## 阶段二：通知阶段

通知阶段（信息维护），一个时间点有一个哨兵向redis集群成员（master、slave）发布信息（publish sentinel：hello），目的确定redis集群成员是否存活，获取状态后，将信息同步到其他的哨兵中。如图：



## 阶段三：故障转移阶段

* **过程描述：**

1. **在通知阶段**，一个哨兵向主机发送hello信息，发现mater没有响应，此时的主机被认定为flags:SRI\_S\_DOWN:**主观下线。**随即这个哨兵就会在哨兵集群中向其他哨兵同步监控的信息，此时其他的哨兵就会发送SENTINEL is-master-down-by-addr。。。，验证master是否下线，当超过半数以上（此参数在哨兵配置文件中配置）的哨兵认为mater下线，就认定masterflags： SRI\_O\_DOWN：**客观下线。**
2. **在哨兵集群中**认为master是主管下线。内部就会发生选举，从所有哨兵中选出一个leader，来处理故障转移工作。选举过程如下：
   1. 所有监控客观下线Master的Sentinel都有可能成为领头Sentinel。
   2. 每次进行领头Sentinel选举之后，不论是否选举成功，所有Sentinel的配置纪元（竞选次数configuration epoch）的值都会自动增加一次。
   3. 在一个配置纪元（竞选次数）里面，所有Sentinel都有一次将某个Sentinel设置为局部领头Sentinel的机会，并且局部领头Sentinel一旦设置，在这个配置纪元里面将不能再更改。
   4. 监视Master客观下线的所有在线Sentinel都有要求其它Sentinel将自己设置为局部领头Sentinel的机会。
   5. 当一个Sentinel（源Sentinel）向另一个Sentinel（目标Sentinel）发送SENTINEL is-master-down-by-addr命令，并且命令中的runid参数不是“.”符号而是当前Sentinel的运行ID时，这表示当前Sentinel要求目标Sentinel将自己设置为领头Sentinel。
   6. Sentinel设置局部领头Sentinel的规则是先到先得。即最先向目标Sentinel发送设置要求的Sentinel将会成为局部领头Sentinel，之后接受到的请求都会被拒绝。
   7. 目标Sentinel接收到SENTINEL is-master-down-by-addr命令后，将向源Sentinel返回一条命令回复，回复中的leader\_runid参数和leader\_epoch参数分别记录了目标Sentinel的局部领头Sentinel的运行ID和配置纪元。
   8. 源Sentinel在接收到目标Sentinel返回的命令回复之后，会检查回复中leader\_epoch参数的值和自己的配置纪元是否相同，如果相同的话，那么源Sentinel继续取出回复中的leader\_runid参数，如果leader\_runid参数的值和源Sentinel的运行ID一直，那么表示目标Sentinel将源Sentinel设置成了局部领头Sentinel。
   9. 如果有某个Sentinel被半数以上的Sentinel设置成了局部领头Sentinel，那么这个Sentinel称为领头Sentinel。
   10. 领头Sentinel的产生需要半数以上的Sentinel支持，并且每个Sentinel在每个配置纪元里面只能设置一次局部Sentinel，所以在一个配置纪元里面，只会出现一个领头Sentinel。
   11. 如果在给定时限内，没有一个Sentinel被选举为领头Sentinel，那么各个Sentinel将在一段时间之后再次进行选举，直到选出领头Sentinel为止。
3. **Sentinel-leader在slave选举出新的**master，选举规则如下：
   1. 删除salve列表中所有处于下线或者短线状态的Slave。
   2. 删除列表中所有最近5s内没有回复过领头Sentinel的INFO命令的Slave。
   3. 删除所有与下线Master连接断开超过down-after-milliseconds \* 10毫秒的Slave。
   4. 领头Sentinel将根据Slave**优先级**，对列表中剩余的Slave进行排序，并选出其中优先级最高的Slave。如果有多个具有相同优先级的Slave，那么领头Sentinel将按照Slave复制**偏移量**，选出其中偏移量最大的Slave。如果有多个优先级最高，偏移量最大的Slave，那么根据**运行ID**最小原则选出新的Master。
   5. 确定新的Master之后，领头Sentinel会以每秒一次的频率向新的Master发送SLAVEOF no one命令，当得到确切的回复role由slave变为master之后，当前服务器顺利升级为Master服务器。
4. **当选出新的Master服务器后**，领头Sentinel会让之前下线Master的Slave发送SLAVEOF命令，让它们复制新的Master。
5. **让已下线的Master服务**器变成新的Master服务器的Slave。当已下线的服务器在此上线后将复新的Master的数据。

# 集群

## 集群简介

1. 业务发展过程中遇到的峰值瓶颈

* redis提供的服务OPS可以达到10万/秒，当前业务OPS已经达到10万/秒
* 内存单机容量达到256G，当前业务需求内存容量1T

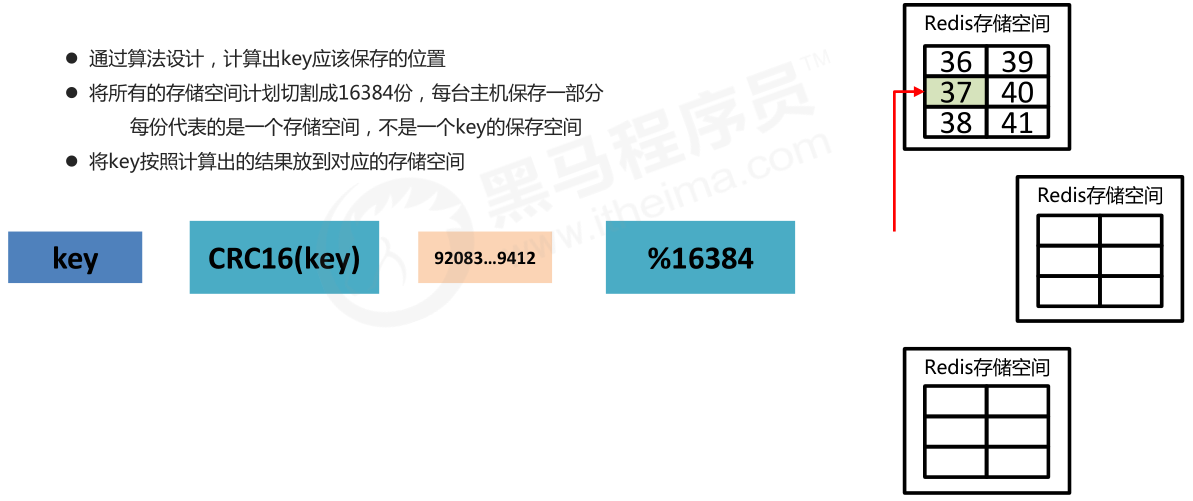
1. 针对以上问题，我们可以使用集群来解决
2. 集群就是使用网络将若干计算机联通起来，并提供统一的管理方式，使其对外呈现单机的服务效果
3. 集群作用

* 分散单台服务器的访问压力，实现负载均衡
* 分散单台服务器的存储压力，实现可扩展性
* 降低单台服务器宕机带来的业务灾难

## 集群结构设计

1. 数据存储设计

* 将所有的存储空间计划切割成16384份（槽），每台主机保存一部分槽位。注意：每个槽位代表是一个存储空间，不是一个key的保存空间。
* 通过算法设计，计算出key应该保存的位置（槽位）
* 按照计算的结果放到对应的存储空间



1. 动态扩展存储空间，数据存储出现如下变化：

* 增加物理存储空间，redis集群会从每个存储空间取出相应数量的槽位放入新的存储空间中。这样就解决了数据存储的问题。
* 减少物理存储空间，redis集群就会将释放出来的槽位，分配到现有的存储空间中

1. 集群内部通讯设计

* 各个Redis存储空间相互通信，保存各个Redis存储空间的槽位的编号数据。
* 当进行数据访问时，通过key计算槽位编号，访问其中一个存储空间，如果一次命中，直接返回，如果一次未命中，就告知具体位置，即一次数据访问，最多2次就能获取数据

## cluster集群搭建

原生安装（单条命令），此次搭建集群为3主3从

1. 配置cluster的redis配置文件，主要参数如下：

* 添加节点

cluster-enabled yes|no

* cluster配置文件名，该文件属于自动生成，仅用于快速查找文件并查询文件内容

cluster-config-file <filename>

* 节点服务响应超时时间，用于判定该节点是否下线或切换为从节点

cluster-node-timeout <milliseconds>

* master连接的slave最小数量

cluster-migration-barrier <count>

* 配置文件如下



1. 启动6台redis

./redis-server ./cluster-conf/redis-6384.conf

1. 检查服务器的ruby的版本及gem的版本,要保持一致

ruby –v

gem –v

1. 使用redis的redis-trib.rb的指令，为了方便，在redis/bin目录下创建软连接

ln –s /usr/local/download/redis-5.0.3/src/redis-trib.rb /usr/local/redis/bin

1. 执行redis命令redis-trib.rb（replicas 1，表示1个master 1个slave）

./redis-cli --cluster create --cluster-replicas 1 -a neusoft 127.0.0.1:6379 127.0.0.1:6380 127.0.0.1:6381 127.0.0.1:6382 127.0.0.1:6383 127.0.0.1:6384

出现如下信息

|  |
| --- |
| Master[0] -> Slots 0 - 5460  Master[1] -> Slots 5461 - 10922  Master[2] -> Slots 10923 - 16383  Adding replica 127.0.0.1:6382 to 127.0.0.1:6379  Adding replica 127.0.0.1:6383 to 127.0.0.1:6380  Adding replica 127.0.0.1:6384 to 127.0.0.1:6381 |

1. redis-cli –c 操作客户端存取数据

./redis-cli -a neusoft -p 6381 –c

1. 测试主从切换

集群内部实现自动主从切换，但是master宕机重新启动时，就变成了slave。

当slave的master宕机后，尝试重连，1s中重连一次。重连时间可以在配置文件中配置cluster-node-timeout <milliseconds>

## Cluster节点操作命令

* 查看集群节点信息

cluster nodes

* 进入一个从节点 redis，切换其主节点

cluster replicate <master-id>

* 发现一个新节点，新增主节点

cluster meet ip:port

* 忽略一个没有solt的节点

cluster forget <id>

* 手动故障转移

cluster failover

# 企业级解决方案

## 缓存预热

1. 现象

Redis服务器短时间感觉卡顿，并且数据库宕机，服务器启动后又迅速宕机

1. 问题排查

* 请求数量较高
* 主从之间数据吞吐量较大，数据同步操作频度较高

1. 解决方案-前置准备工作：
2. 日常例行统计数据访问记录，统计访问频度较高的热点数据
3. 利用LRU数据删除策略，构建数据留存队列，例如：storm与kafka配合
4. 解决方案-准备工作
5. 将统计结果中的数据分类，根据级别，redis优先加载级别较高的热点数据
6. 利用分布式多服务器同时进行数据读取，提速数据加载过程
7. 热点数据主从同时预热
8. 解决方案-实施
9. 使用脚本程序固定触发数据预热过程
10. 如果条件允许，使用了CDN（内容分发网络），效果会更好
11. 总结

缓存预热就是系统启动前，提前将相关的缓存数据直接加载到缓存系统。避免在用户请求的时候，先查询数据库，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！