1. redis概述

1.1 简介

- 截止到2021年12月 数据库排名https://db-engines.com/en/ranking

381 systems in ranking, December 2021

Rank			DBMS	Database Model	Score		
Dec 2021	Nov 2021	Dec 2020			Dec 2021	Nov 2021	Dec 2020
1.	1.	1.	Oracle	Relational, Multi-model	1281.74	+9.01	-43.86
2.	2.	2.	MySQL	Relational, Multi-model	1206.04	-5.48	-49.41
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	954.02	-0.27	-84.07
4.	4.	4.	PostgreSQL	Relational, Multi-model	608.21	+10.94	+60.64
5.	5.	5.	MongoDB	Document, Multi-model	484.67	-2.67	+26.95
6.	6.	↑ 7.	Redis	Key-value, Multi-model	173.54	+2.04	+19.91
7.	7.	↓ 6.	IBM Db2	Relational, Multi-model	167.18	-0.34	+6.74
8.	8.	8.	Elasticsearch	Search engine, Multi-model	157.72	-1.36	+5.23
9.	9.	9.	SQLite	Relational	128.68	-1.12	+7.00
10.	↑ 11.	↑ 11.	Microsoft Access	Relational	125.99	+6.75	+9.25

- redis(Remote Dictionary Server) 一个开源的key-value存储系统
- 它支持存储的Value类型：包括String(字符串),list(链表),set(集合),zset(sorted set 有序集合),hash(哈希类型)。都支持push/pop、add/romove，获取交集、并集、差集等一些相关操作，操作是原子性的。
- redis支持各种不同方式的排序
- redis (与memcatched相同)数据存在内存中
- redis会周期性的把更新的数据写入磁盘，或者把修改的操作追加到记录文件
- redis支持集群，实现master-slave(主从)同步操作

1.2 应用场景

缓存：配合关系型数据库做高速缓存

计数器：进行自增自减运算

时效性数据：利用expire过期，例如手机验证码功能

海量数据统计：利用位图，存储用户是否是会员、日活统计、文章已读统计、是否参加过某次活动

会话缓存：使用redis统一存储多台服务器用到的session信息

分布式队列/阻塞队列：通过List双向链表实现读取和阻塞队列

分布式锁: 使用redis自带setnx命令实现分布式锁

热点数据存储：最新文章、最新评论，可以使用redis的list存储，ltrim取出热点数据，删除旧数据

社交系统：通过Set功能实现，交集、并集实现获取共同好友，差集实现好友推荐，文章推荐

排行榜：利用sorted-set的有序性，实现排行榜功能，取top n

延迟队列：利用消费者和生产者模式实现延迟队列

去重复数据：利用Set集合，去除大量重复数据

发布/订阅消息：pub/sub模式

2.Redis安装

2.1 前置处理环境

VMware安装

安装centOS的linux操作系统

xshell

xftp

2.2 配置虚拟机网络

按ctrl+alt+f2 切换到命令行

cd (/)目录

修改/etc/sysconfig/network-scripts/ifcfg-ens33

vi 命令

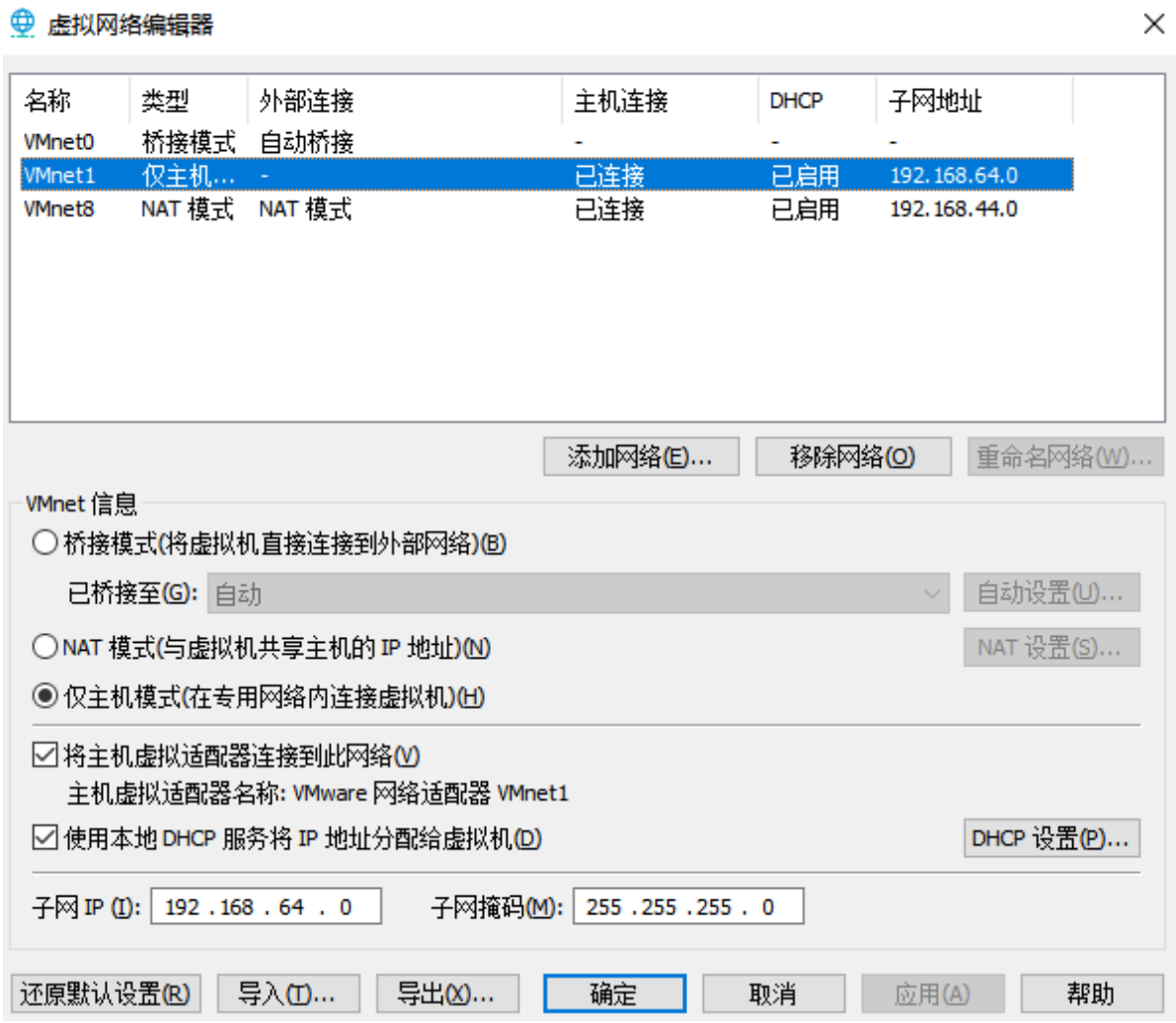
按insert表示插入

按ctrl+esc退出修改状态

:wq 写入并退出

```
TYPE=Ethernet
PROXY_METHOD=none
BROWSER_ONLY=no
BOOTPROTO=static
DEFROUTE=yes
IPV4_FAILURE_FATAL=no
IPV6INIT=yes
IPV6_AUTOCONF=yes
IPV6_DEFROUTE=yes
IPV6_FAILURE_FATAL=no
IPV6_ADDR_GEN_MODE=stable-privacy
NAME=ens33
UUID=49f946c2-228c-4535-b976-c7c67f7f6907
DEVICE=ens33
ONBOOT=yes

GATEWAY=192.168.44.2
IPADDR=192.168.44.4
NETMASK=255.255.255.0
DNS1=8.8.8.8
DNS2=8.8.8.4
```

vmware 菜单 编辑->虚拟网络编辑器


虚拟网络编辑器
名称 类型 外部连接 主机连接 DHCP 子网地址
VMnet0 桥接模式 自动桥接 - - -
VMnet1 仅主机... - 已连接 已启用 192.168.64.0
VMnet8 NAT 模式 NAT 模式 已连接 已启用 192.168.44.0
添加网络(E)... 移除网络(O) 重命名网络(W)...
VMnet 信息
桥接模式(将虚拟机直接连接到外部网络)(B)
已桥接至(G): 自动 自动设置(U)...
NAT 模式(与虚拟机共享主机的 IP 地址)(N) NAT 设置(S)...
仅主机模式(在专用网络内连接虚拟机)(H)
将主机虚拟适配器连接到此网络(V)
主机虚拟适配器名称: VMware 网络适配器 VMnet1
使用本地 DHCP 服务将 IP 地址分配给虚拟机(D) DHCP 设置(P)...
子网 IP (I): 192 . 168 . 64 . 0 子网掩码(M): 255 . 255 . 255 . 0
还原默认设置(R) 导入(T)... 导出(X)... 确定 取消 应用(A) 帮助


虚拟网络编辑器
名称 类型 外部连接 主机连接 DHCP 子网地址
VMnet0 桥接模式 自动桥接 - - -
VMnet1 仅主机... - 已连接 已启用 192.168.64.0
VMnet8 NAT 模式 NAT 模式 已连接 已启用 192.168.44.0
添加网络(E)... 移除网络(O) 重命名网络(W)...
VMnet 信息
桥接模式(将虚拟机直接连接到外部网络)(B)
已桥接至(G): 自动 自动设置(U)...
NAT 模式(与虚拟机共享主机的 IP 地址)(N) NAT 设置(S)...
仅主机模式(在专用网络内连接虚拟机)(H)
将主机虚拟适配器连接到此网络(V)
主机虚拟适配器名称: VMware 网络适配器 VMnet8
使用本地 DHCP 服务将 IP 地址分配给虚拟机(D) DHCP 设置(P)...
子网 IP (I): 192 . 168 . 44 . 0 子网掩码(M): 255 . 255 . 255 . 0
还原默认设置(R) 导入(T)... 导出(X)... 确定 取消 应用(A) 帮助

NAT设置

2.3 下载地址

https://download.redis.io/releases/redis-6.2.6.tar.gz 稳定版

2.4 安装

2.4.1 上传文件

使用xftp 实现对文件上传到虚拟机opt目录下

新建会话属性

常规 选项

站点

名称(N): 192.168.44.4

主机(H): 192.168.44.4

协议(R): SFTP 设置(S)...

端口号(O): 22

代理服务器(X): <无> 浏览(W)...

说明(D):

登录

匿名登录(A)

使用身份验证代理(G)

方法(M):

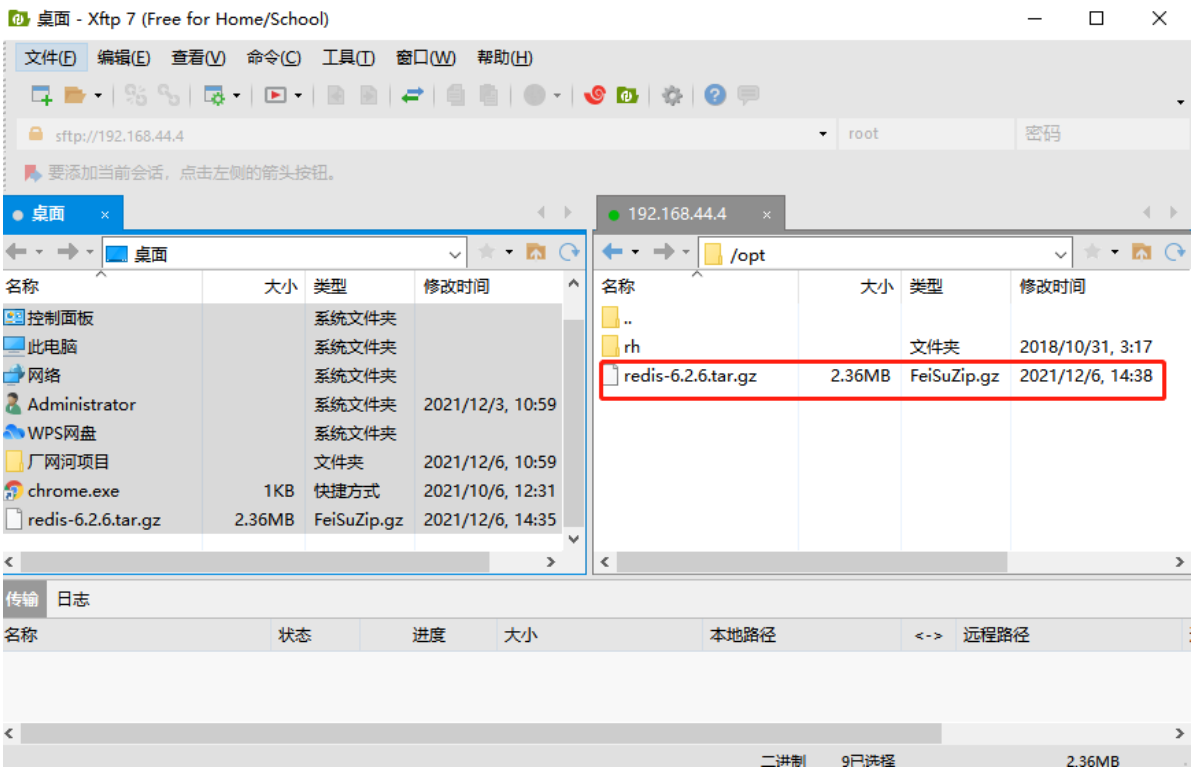
- [x] Password
- [ ] Public Key
- [ ] Keyboard Interactive
- [ ] GSSAPI
- [ ] PKCS11
- [ ] CAPI

设置(T)... 上移(V) 下へ(E)

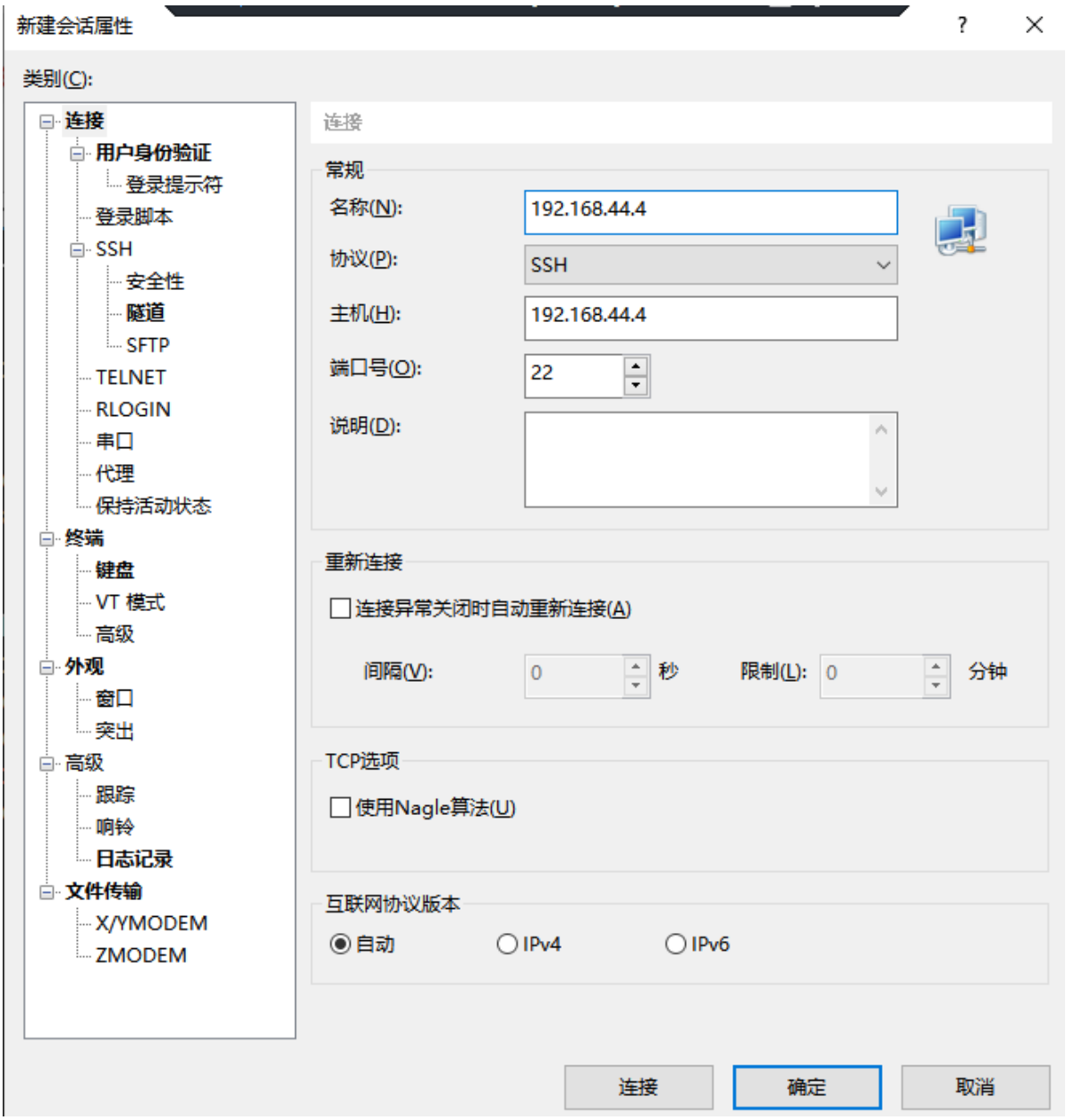
用户名(U): root

密码(P): ••••••••••••

连接 确定 取消



2.4.2 使用xshell连接到虚拟机



2.4.3 检测安装gcc

- 检测 gcc -v
- 安装 yum install gcc

提示：ctrl+l 清屏

2.4.4 解压redis文件

tar -zxvf redis-6.2.6.tar.gz

2.4.5 编译redis

进入/opt/redis-6.2.6/

运行make 编译

2.4.6 安装redis

make install

自动安装到/usr/local/bin目录下

2.4.7 安装后文件概述

redis-benchmark：性能测试工具

redis-check-aof：修复aof持久化文件

redis-check-rdb：修复rdb持久化文件

redis-cli：redis命令行工具

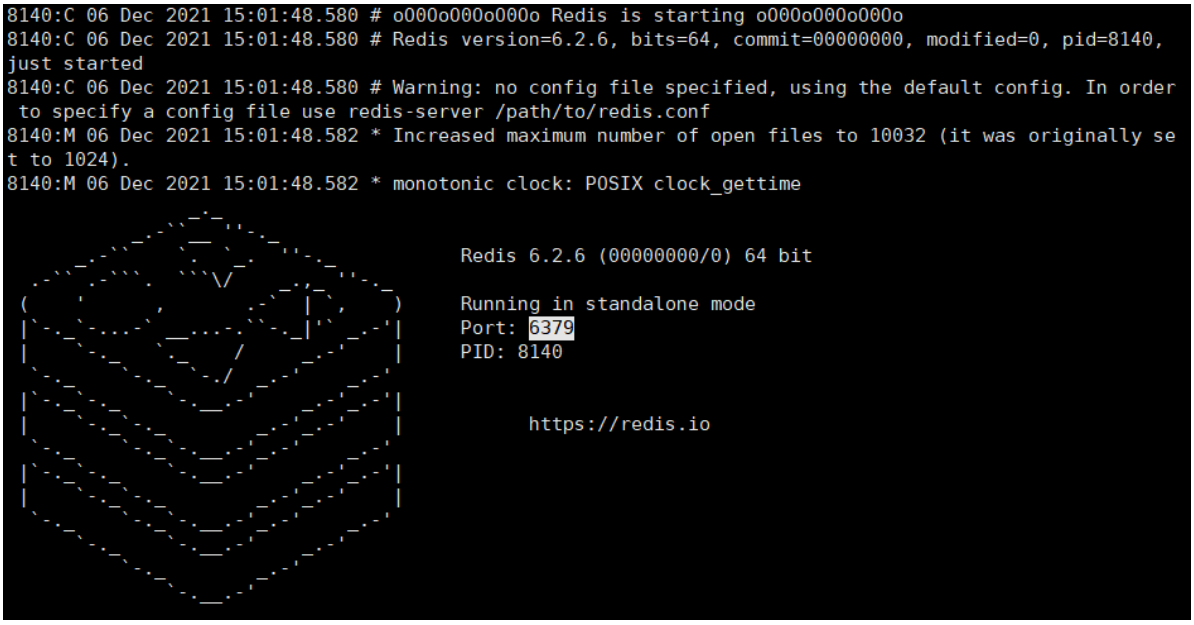
redis-sentinel：redis集群哨兵使用

redis-server：启动redis

3.redis启动

3.1 前后启动(不推荐)

调用redis-server，启动后xshell窗口不能再做其他操作，ctrl+c退出



3.2 后台启动(推荐)

1. 进入/opt/redis-6.2.6下，找到redis.conf文件，通过 cp redis.conf redis_1.conf进行复制
2. 使用vi 编辑redis_1.conf文件，将daemonize的no改成yes，允许后台启动(vi 查找 / 查找内容，向下n。按insert在插入模式下修改 :wq保存退出)
3. 启动redis redis-server redis_1.conf
4. 查看redis进程 ps -ef|grep redis

3.3 启动命令行

redis-cli

显示`127.0.0.1:6379>`

表示redis在6379端口启动成功

输入ping命令测试

```
127.0.0.1:6379> ping
PONG
```

3.4 退出redis

1.在redis-cli调用shutdown, 再执行exit

2.先执行exit，再杀死reids进程 kill -9 redis进程id

4.redis使用的基本命令

1.默认16个数据库，类似数组下标从0开始，初始默认使用0号库。使用select 命令进行切换。语法

select < dbid>

select 1

2.统一密码管理，所有库使用同样的密码

3.dbsize查看当前数据库的key的数量

4.flushdb 清空当前库

5.flushall 清空全部库

6.keys * 查看当前库所有key

7.exists < key>判断某个key是否存在

8.type < key> 相看key的类型

9.object encoding < key> 查看底层数据类型

10.del < key>删除指定的key数据

11.unlink < key> 根据选择非阻塞删除。仅将key从keyspace元数据中删除，真正的删除会在后续中做异步操作

12.expire < key>< seconds>：为给定的key设置过期时间，以秒为单位

13.ttl < key>：查看给定key的过期时间：-1表示永不过期 -2 表示已过期

5.redis常用的五种数据类型

5.1 Redis String字符串

5.1.1 简介

- String类型在redis中最常见的一种类型
- string类型是二制安全的，可以存放字符串、数值、json、图像数据
- value存储最大数据量是512M

5.1.2 常用命令

- **set** < key>< value>：添加键值对
  - nx：当数据库中key不存在时，可以将key-value添加到数据库
  - xx: 当数据库key存在时，可以将key-value添加到数据库，与nx参数互斥
  - ex: 设置key-value添加到数据库，并设置key的超时时间(以秒钟为单位)
  - px:设置key-value添加到数据库，并设置key的超时时间(以豪秒钟为单位)，与ex互斥
- **get** < key>查询对应键值
- **append** < key>< value>：将给定的值追加到key的末尾
- **strlen** < key>：获取值的长度
- **setnx** < key>< value>：只有在key不存在时，设置key-value加入到数据库
- **setex** < key> < timeout>< value>：添加键值对，同时设置过期时间(以秒为单位)
- **incr** < key>：将key中存储的数字加1处理，只能对数字值操作。如果是空，值为1
- **decr** < key>：将key中存储的数字减1处理，只能对数字值操作。如果是空，值为1
- **incrby** < key>< increment>：将key中存储的数字值增加指定步长的数值,如果是空，值为步长。(具有原子性)
- **decrby** < key>< decrement>: 将key中存储的数字值减少指定步长的数值,如果是空，值为步长。(具有原子性)
- **mset** < key1>< value1>[< key2>< value2>...]：同时设置1个或多个key-value值
- **mget** < key1>[< key2>...]：同时获取1个或多个value
- **msetnx** < key1>< value1>[< key2>< value2>...]：当所有给定的key都不存在时，同时设置1个或多个key-value值(具有原子性)
- **getrange/substr** < key>< start>< end> 将给定key，获取从start(包含)到end(包含)的值
- **setrange** < key>< offset>< value>：从偏移量offset开始，用value去覆盖key中存储的字符串值
- **getset**< key>< value>： 对给定的key设置新值，同时返回旧值。如果key不存在，则添加一个key-value值

5.1.3应用场景

单值缓存 set key value　　get key

对象缓存

set stu:001 value(json)

mset stu:001:name zhangsan　stu:001:age 18　stu:001:gender 男

mget　stu:001:name　stu:001:age

分布式锁

setnx key:001 true //返回1代表加锁成功

setnx key:001 true //返回0代表加锁失败

//.....业务操作

del key:001 //执行完业务释放锁

set key:001 true ex 20 nx //防止程序意外终止导致死锁

计数器

incr article:read:1001 //统计文章阅读数量

分布式系统全局序列号

incrby orderid 100 //批量生成序列号

5.2 Redis List列表

5.2.1 简介

- Redis列表是简单的字符串列表，单键多值。按照插入顺序排序。可以添加一个元素到列表的头部(左边)或者尾部(右边)
- 一个列表最多可以包含2的32次方-1个元素
- 底层是一个双向链表，对两端的操作性能很高，通过索引下标的操作中间的节点性能会较差

5.2.2 常用命令

- lpush < key> < value1>[< value2>...]：从左侧插入一个或多个值
- lpushx < key> < value1>[< value2>...]：将一个或多个值插入到已存在的列表头部
- lrange < key>< start>< stop>：获取列表指定范围内的元素，0左边第1位，-1右边第1 位，0 ~-1取出所有
- rpush < key> < value1>[< value2>...]：从右侧插入一个或多个值
- rpushx < key> < value1>[< value2>...]：将一个或多个值插入到已存在的列表尾部
- lpop < key>[count]：移除并获取列表中左边第1个元素，count表明获取的总数量,返回的为移除的元素
- rpop < key>[count]：移除并获取列表中右边第1个元素，count表明获取的总数量,返回的为移除的元素
- rpoplpush < source>< destination>：移除源列表的尾部的元素(右边第一个)，将该元素添加到目标列表的头部(左边第一个)，并返回
- lindex < key>< index>:通过索引获取列表中的元素
- llen < key>：获取列表长度
- linsert < key> before|after < pivot>< element>：在< pivot>基准元素前或者后面插入< element>，如果key不存在，返回0。如果< pivot>不存在，返回-1，如果操作成功，返回执行后的列表长度
- lrem < key>< count>< element>：根据count的值，移除列表中与参数相等的元素
  - count=0 移除表中所有与参数相等的值
  - count>0 从表头开始向表尾搜索，移除与参数相等的元素，数量为count
  - count<0 从表尾开始向表头搜索，移除与参数相等的元素，数量为count的绝对值
- lset < key>< index> < element>：设置给定索引位置的值

- ltrim< key>< start> < stop>：对列表进行修剪，只保留给定区间的元素，不在指定区间的被删除
- brpop < key> timeout：阻塞式移除指定key的元素，如果key中没有元素，就等待，直到有元素或超时，执行结束。

5.2.3 应用场景

- 数据队列
  - 堆栈stack=lpush+lpop
  - 队列queue=lpush+rpop
  - 阻塞式消息队列 blocking mq=lpush+brpop
- 订阅号时间线

  lrange key start stop

5.3 Redis Hash 哈希

5.3.1 简介

是一个string类型的键和value（对象），特别适合于存储对象，类似于java里面学习的Map<String,Object>。

假设场景：需要在redis中存储学生对象,对象属性包括(id,name,gender,age)，有以下几种处理方式

方式一：用key存储学生id，用value存储序列化之后用户对象(如果用户属性数据需要修改，操作较复杂，开销较大)

方式二：用key存储学生id+属性名，用value存储属性值（用户id数据冗余）

方式三：用key存储学生id，用value存储field+value的hash。通过key(学生d)+field(属性)可以操作对应数据。

5.3.2 常用命令

- hset < key>< field>< value>[< field>< value>...]：用于为哈希表中的字段赋值，如果字段在hash表中存在，则会被覆盖
- hmset:用法同hset，在redis4.0.0中被弃用
- hsetnx < key>< field>< value>：只有在字段不存在时，才设置哈希表字段中的值
- hget < key>< field> 返回哈希表中指定的字段的值
- hmget < key>< field>[< field>...]：获取哈希表中所有给定的字段值
- hgetall < key>：获取在哈希表中指定key的所有字段和值
- hexists < key>< field>：判断哈希表中指定的字段是否存在，存在返回1 ，否则返回0
- hkeys < key>：获取哈希表中所有的字段
- hvals < key>：获取哈希表中所有的值
- hlen < key>：获取哈希表中的field数量
- hdel < key>< field>[< field>...]：删除一个或多个哈希表字段
- hincrby < key>< field>< increment>：为哈希表key中指定的field字段的**整数**值加上增加increment
- hincrbyfloat < key>< field>< increment>：为哈希表key中指定的field字段的**浮点数**值加上增加increment

5.3.3 应用场景

1.对象缓存 hset stu:001 name zhangsan age 20 gender man

2.电商购物车操作

- 以用户id作为key, 以商品id作为field，以商品数量作为value
- 添加商品: hset user:001 s:001 1
- hset user:001 s:002 2
- 增减商品数量:hincrby user:001 s:001 3
- 查看购物车商品总数: hlen user:001
- 删除商品 : hdel user:001 s:001
- 获取所有商品: hgetall user:001

5.4 Redis Set集合

5.4.1 简介

set是string类型元素无序集合。对外提供的功能和list类似，是一个列表功能。集合成员是唯一的。

5.4.2 常用命令

- sadd < key>< member>[< member>...]：将一个或多个成员元素加入到集合中，如果集合中已经包含成员元素，则被忽略
- smembers < key>：返回集合中的所有成员。
- sismember < key>< member>：判断给定的成员元素是否是集合中的成员，如果是返回1,否则返回0
- scard < key>：返回集合中元素个数
- srem < key>< member>[< member>...]：移除集合中一个或多个元素
- spop < key>[< count>]：移除并返回集合中的一个或count个随机元素
- srandmember < key>[< count>]：与spop相似，返回随机元素，不做移除
- smove < source> < destination> < member>：将member元素从source源移动到destination目标
- sinter < key>[< key>...]：返回给定集合的交集(共同包含)元素
- sinterstore < destination> < key1>[< key2>...]：返回给定所有集合的交集，并存储到destination目标中
- sunion < key>[< key>...]：返回给定集合的并集(所有)元素
- sunionstore < destination> < key1>[< key2>...]：返回给定所有集合的并集，并存储到destination目标中
- sdiff < key>[< key>...]：返回给定集合的差集(key1中不包含key2中的元素)
- sdiffstore < destination> < key1>[< key2>...]：返回给定所有集合的差集，并存储到destination目标中

5.4.3 应用场景

- 抽奖
  - 参与抽奖：sadd cj001 user:13000000000 user:13455556666 user:13566667777
  - 查看所有参与用户： smembers cj001
  - 实现抽奖： spop cj001 3 / srandmember cj001 3
- 朋友圈点赞 快手/抖音
  - 点赞 sadd like:friend001 user:001

    sadd like:friend001 user:002

    - 取消点赞　srem like:friend001 user:001
    - 判断用户是否已点赞 sismember like:friend001 user:001
    - 显示点赞用户　smembers like:friend001
    - 获取点赞次数 scard like:friend001
- 关注模型：sinter交集　sunion并集　sdiff 差集
    - 微博　sadd g:list:u001 1001　sadd g:list:u002 1001　你们共同关注的 sinter交集
    - QQ 你们有共同好友 sinter交集
    - 快手 可能认识的人 sdiff差集

5.5 Redis zset有序集合

5.5.1 简介

- 有序集合是string类型元素的集合，不允许重复出现成员
- 每个元素会关联一个double类型的分数，redis是通过分数为集合中的成员进行从小到大的排序
- 有序集合的成员是唯一的，但是分数可以重复
- 成员因为有序，可以根据分数或者次序来快速获取一个范围内的元素

5.5.2常用命令

- zadd < key> < score>< member>[< score>< member>...]：将一个或多个元素及其分数加入到有序集合中
- zrange < key>< min>< max> [byscore|bylex] [rev] [limit offset count] [withscores]：返回有序集合指定区间的成员，（byscore按分数区间，bylex按字典区间，rev 反向排序(分数大的写前边，小的写后边)，limit分页(offset偏移量，count返回的总数)，withscores返回时带有对应的分数)
- zrevrange < key>< start>< stop>[limit offset count]：返回集合反转后的成员
- zrangebyscore < key>< min>< max> [withscores] [limit offset count]：参考zrange用法
- zrevrangebyscore< key>< max>< min> [withscores] [limit offset count]：参考zrange用法
- zrangebylex < key>< min>< max>] [limit offset count]：通过字典区间返回有序集合的成员
- zrangebylex k2 - +：减号最小值,加号最大值
- zrangebylex k2 [aa (ac：[中括号表示包含给定值，(小括号表示不包含给定值
- zcard < key>：获取集合中的成员数量
- zincrby < key> < increment>< member> ：为集合中指定成员分数加上增量increment
- zrem < key> < member>[< member>...]：移除集合的一个或多个成员
- zcount < key>< min>< max>：统计集合中指定区间分数(都包含)的成员数量
- zrank < key>< member>：获取集合中成员的索引位置
- zscore < key>< member>：获取集合中成员的分数值

5.3.3 应用场景

1.按时间先后顺序排序：朋友圈点赞 zadd 1656667779666

2.热搜　微博 今日头条　快手

3.获取topN　zrevrange k1 300 10 limit 0 10

6.Redis AOF

6.1简介

目前，redis的持久化主要应用AOF(Append Only File)和RDB两大机制。AOF以日志的形式来记录每个写操作(增量保存),将redis执行过的所有写指令全部记录下来(读操作不记录)。只许追加文件，但不可以改写文件。redis启动之初会读取该文件，进行重新构建数据。

6.2 AOF的配置

- AOF默认不开启,在conf配置文件中进行配置。
- 修改redis.conf配置文件

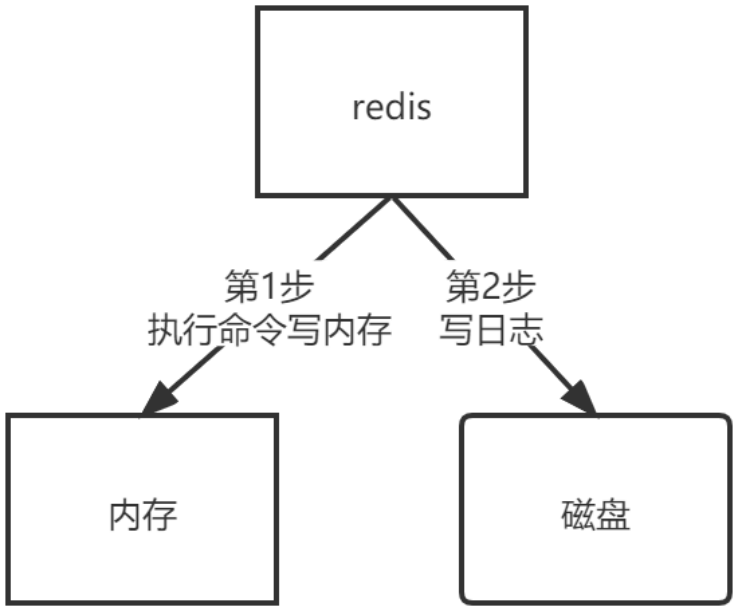
```
appendonly no
//修改
appendonly yes
```

- 默认文件名是appendonly.aof
- 默认是启动后的相对路径，redis在哪里启动，appendonly.aof文件就在哪生成

6.3 AOF日志是如何实现

数据库写前日志(Write Ahead Log ,WAL)，在实际写数据库前，先把修改的数据记录到日志文件中，以便发生故障时，时行恢复。

AOF日志是写后日志。redis先去执行命令，把数据写入内存中，然后才去记录日志。



查看AOF文件

set k1 v1

vi appendonly.aof

```

```

```
*3     //接下来的指令由3部分组成

$3    //指令有3个字节

set

$2   //指令有2个字节

k1

$2 //指令有2个字节

v1
```

为什么使用写后日志?

1.redis为了避免检查开销，向AOF中记录日志，是不做检查的。如果写前执行，很有可能将错误指令记录到日志中，在使用redis恢复日志时，就可能会出现错误

2.不会阻塞当前的写操作

6.4 AOF潜在风险

1.aof文件可能由于异常原因被损坏。可以使用redis自带的命令redis-check-aof --fix appendonly.aof文件，修复成功，可以正确启动

2.由于刚刚执行一个指令，还没有写入日志，就宕机了。就会导致数据永久丢失(redis做为数据库存储的情况)

3.AOF避免了对当前指令的阻塞，但可能会由于磁盘写入压力较大，对下一个操作带来阻塞风险

6.5 AOF三种写回策略

打开redis.conf配置文件　appendfsync选项

always：同步写回:每个写指令执行完，立即同步将指令写入磁盘日志文件中

everysec：每秒写回：默认配置方式。每个写指令执行完，先把日志写到AOF文件的内存缓冲区。每隔一秒把缓冲区的内容写入磁盘

no：操作系统控制写回：每个写指令执行完，先把日志写到AOF文件的内存缓冲区，由操作系统决定何时把缓冲区的内容写入磁盘

选项	写日志时机	优点	缺点
always	同步写回	数据可靠性高，基本不丢失	对性能影响较大
everysec	每秒写回	性能适中	当服务器宕机时，丢失上1秒内的数据
no	操作系统控制写回	性能最高	当服务器宕机时，丢失的数据较多

6.6 AOF重写机制

6.6.1 简介

Redis根据数据库现有数据，创建一个新的AOF文件，读取数据库中所有键值对，重新对应一条命令写入。

可以使用命令bgrewriteaof

6.6.2 AOF重写的相关配置

```
auto-aof-rewrite-percentage 100
auto-aof-rewrite-min-size 64mb
```

如果aof文件超过64m，且比上次重写后的大小增加了100%，自动触发重写。

例如　文件80m，开如重写，重写后降到50m，下一次，达到100m再开始重写。

6.6.3 AOF重写流程

- bgrewirteaof触发重写，判断是否当前有重写在运行，如果有，则等待重写结束后再执行
- 主进程fork出一个子进程，执行重写操作，保证主进程不阻塞，可以继续执行命令
- 子进程循环遍历reids内存中的所有数据到临时文件，客户端的写请求同时写入aof缓冲区和aof重写缓冲区。保证原AOF文件完整以及新的AOF文件生成期间的新的数据修改操作不会丢失
- 子进程写完新AOF文件以后，向主进程发送信号，主进程更新统计信息
- 主进程把aof重写缓冲区中的数据写入到新的AOF文件
- 用新AOF文件覆盖掉旧的AOF文件，完成AOF重写

7.Redis RDB

7.1简介

RDB:Redis DataBase的缩写。内存快照，记录内存中某一时刻数据的状态。

RDB和AOF相比，记录的数据，不是操作指令。

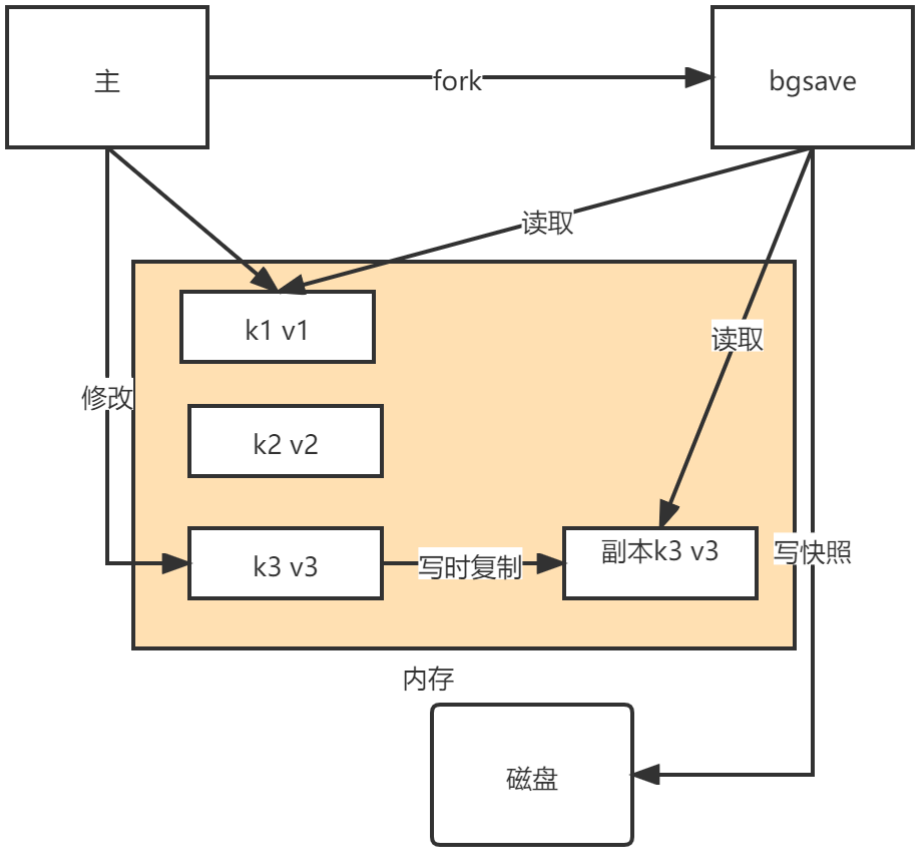
redis提供了两个命令生成RDB文件

save:在主线程中执行，会导致阻塞

bgsave:创建一个子进程，专门用来写RDB，避免主线程的阻塞，默认配置。

例：6GB内存数据量，磁盘的写入0.3GB/S，需要20S时间，来完成RDB文件的写入。

处理技术：写时复制技术（copy-on-write cow)。在执行快照处理时，仍然正确执行写入操作



7.2 快照频率

通过redis.conf配置文件去做处理

```
# save 3600 1
# save 300 100
# save 60 10000
```

7.3 混合使用AOF和RDB

通过redis.conf配置文件

打开aof

```
appendonly yes
```

打开混合配置

```
aof-use-rdb-preamble yes
```

混合过程

```
set k1 v1
set k2 v2
set k3 v3
bgrewriteaof
set k4 v4
set k5 v5
```

在aof文件中，前半部分，就是rdb文件的内容，从rewirte之后，是aof文件内容

7.4关于对redis执久化处理的建议

- 如果数据在服务器运行的时候，使用redis做缓冲，可以不使用任何持久化方式
- 数据不能丢失，rdb和aof混合使用是一个好的选择
- 如果数据不要求非常严格，要以允许分钟级别丢失，可以使用rdb
- 如果只使用AOF，建议配置策略是everysec，在可靠性和性能之间做了一个折中
- 如果磁盘允许，尽量避免AOF重写的频率,将默认值64M进行修改

8.主从复制

8.1 简介

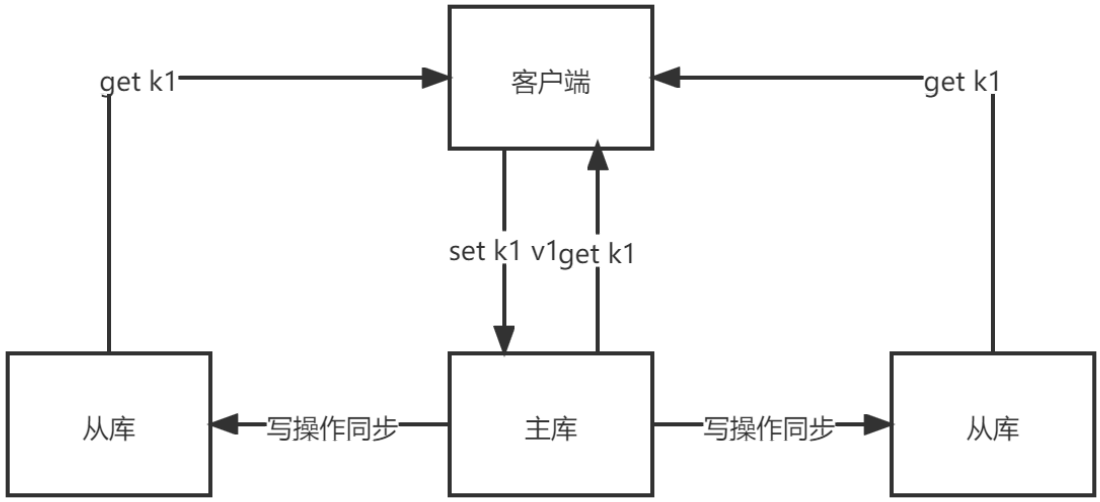
主从库之间采用读写分离的方式

读操作: 主库、从库都可以处理

写操作：首先写到主库执行，然后再将主库同步给从库。

实现读写分离，性能扩展

容灾快速恢复



8.2 主从复制步骤

1.创建一个目录 ,在root下创建一个myredis的目录

```
mkdir myredis//创建目录
cd myredis //进行目录
cp /opt/reids-6.2.6/redis.conf redis_1.conf  //cp 源文件 目标文件
ls
```

2.关闭aof

3.使用vi编辑三个conf文件，redis_6379.conf,redis_6380.conf,redis_6381.conf做为一主二从配置

```
include redis_1.conf
pidfile /var/run/reids_6379.pid
port 6379
dbfilename dump6379.rdb
```

4.分别启动三个服务

```
[root@localhost myredis]# redis-server redis_6379.conf
[root@localhost myredis]# redis-server redis_6380.conf
[root@localhost myredis]# redis-server redis_6381.conf
[root@localhost myredis]# ps -ef|grep redis
root       7289      1  0 17:35 ?        00:00:00 redis-server 127.0.0.1:6379
root       7296      1  0 17:35 ?        00:00:00 redis-server 127.0.0.1:6380
root       7302      1  0 17:35 ?        00:00:00 redis-server 127.0.0.1:6381
root       7316   6663  0 17:35 pts/0    00:00:00 grep --color=auto redis
```

5.在三个客户端，模拟分别连接到不同服务器

```
redis-cli -p 端口号
redis-cli -p 6379
redis-cli -p 6380
redis-cli -p 6381
```

6.查看服务器状态

```
info replication
```

7.在6380和6381上调用replicaof，将其从属于6379

```
replicaof 127.0.0.1 6379
```

8.在主库上可以写入数据，从库不能写入数据

9.主库和从库都可以读数据

8.3 服务器宕机演示

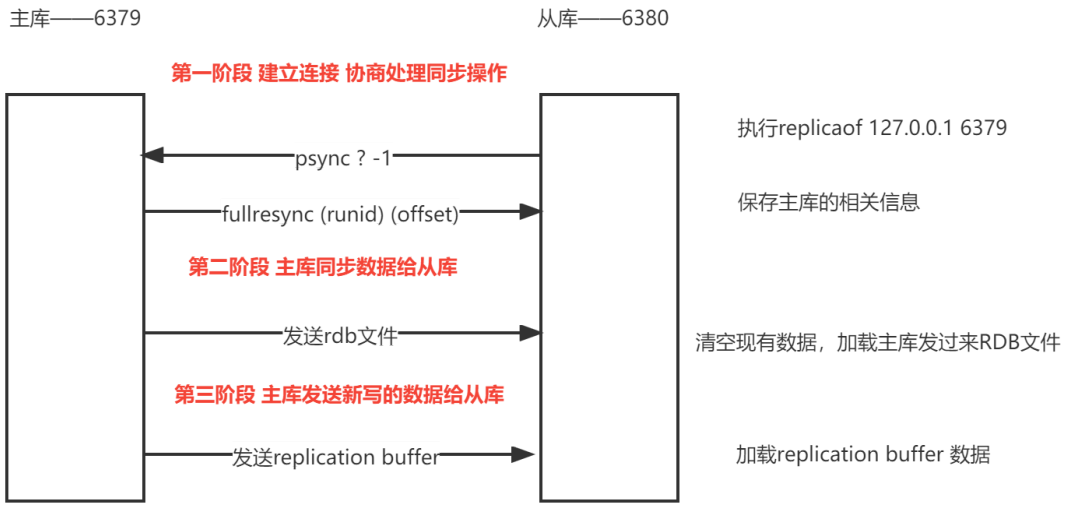
8.3.1 从服务器宕机

- 6381上调用shutdown
- 在主服务器上写入数据
- 6381重新连上时，仍然可以接收到主服务器的数据

8.3.2 主服务器宕机

- 6379服务器调用shutdown
- 在从服务器上仍然可以读取数据
- 从服务器显示主服务器的状态为down
- 当主服务器重新启动，从服务器显示主服务器的状态是up

8.4 主从同步原理



第一阶段，主从建立连接，协商同步。从库和主库建立连接，告诉主库即将进行同步操作。主库需要确认并回复，主从就可以开始进行同步处理了。

从库向主库发送一个psync指令,包含两个参数。一个是主库的runID，另一个是复制进度offset。

- runID是每个redis实例启动时生成的一个随机ID，唯一标识。第一次复制时，从库不知道主库的runid，所以设为一个"?"。
- offset，-1表示第一次复制

主库收到指令后，会发送给从库fullresync指令去响应，带着主库的runid，还有目前复制进度offset。从库会记录下这两个参数。fullresync表示全量复制。主库把所有内容都复制给从库

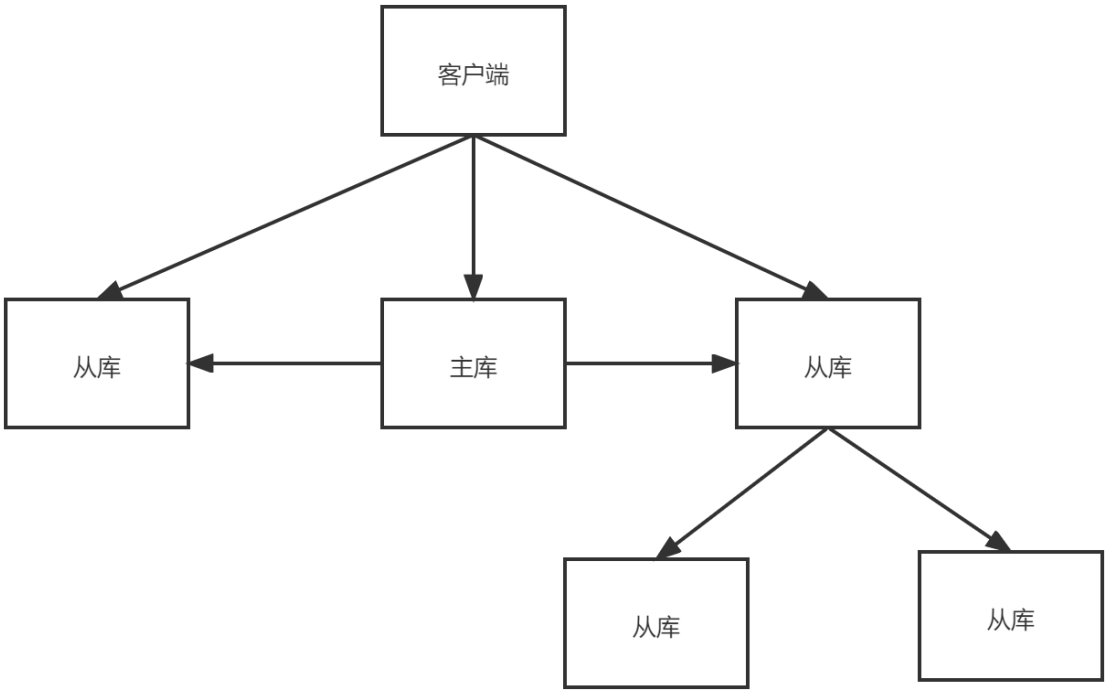
第二阶段，主库将所有数据发送给从库进行同步。从库收到rdb文件后，在本地把原有的数据清除，同步从主库接收到的rdb文件 。

如果在主库把数据跟从库同步的过程中，主库还有数据写入，为了保证主从数据的一致性，主库会在内存中给一块空间replication buffer，专门记录rdb文件生成后收到的所有写操作

第三阶段，主库把第二阶段执行过程中新收到的操作，再发送给从库，从库再加载执行这些操作，就实现同步处理了。

8.5 主-从-从模式

采用主-从-从模式，将主库生成和传输rdb文件的压力，以级联方式分散到从库上。

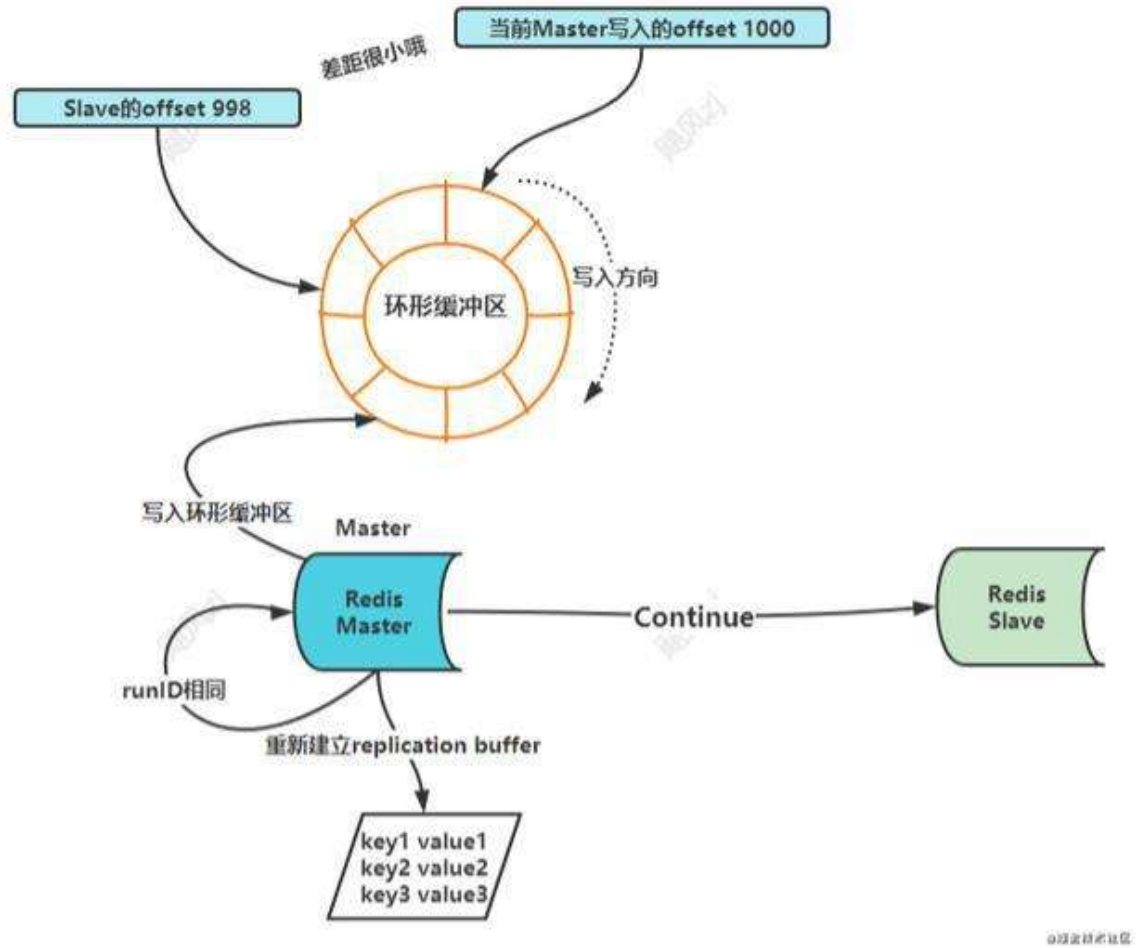


8.6 网络连接异常情况

在redis2.8之前，如果网络异常，再次连接后，需要做全量复制

从redis2.8之后，采用增量复制方式。repl_backlog_buffer缓冲区。当主从网络断开后，主库把收到写操作，写入replication buffer,同时，也写入到repl_backlog_buffer缓冲区.

这个缓冲区，是一个环形缓冲区，主库会记录自己写到的位置，从库会记录自己读到的位置。



repl_backlog_size参数

缓冲空间大小=主库写入速度 * 操作大小-主从库网络传输速度 * 操作大小

repl_backlog_size=缓冲空间大小*2

2000 * 2-1000 * 2=2000 2M   4M

9.哨兵模式

9.1 简介

当主库宕机，在从库中选择一个，切换为主库。

问题:

1、主库是否真正宕机?

2、哪一个从库可以作为主库使用?

3、如何实现将新的主库的信息通过给从库和客户端?

9.2 基本流程

哨兵主要任务:

- 监控
- 选择主库
- 通知

9.3 哨兵模式配置

1.创建一个sentinel.conf文件，进行配置

```
#端口号
port 26379
#sentinel monitor <自定义的reids主节点名称> <IP> <port> <数量>
sentinel monitor mymaster 127.0.0.1 6379 1
#指定多少毫秒后，主节点没有应答哨兵，就认为下线了
sentinel down-after-milliseconds mymaster 30000
```

2.启动三个redis实例，配置成一主二从模式

3.启动哨兵

4.将主服务器宕机，观察哨兵监控信息变化

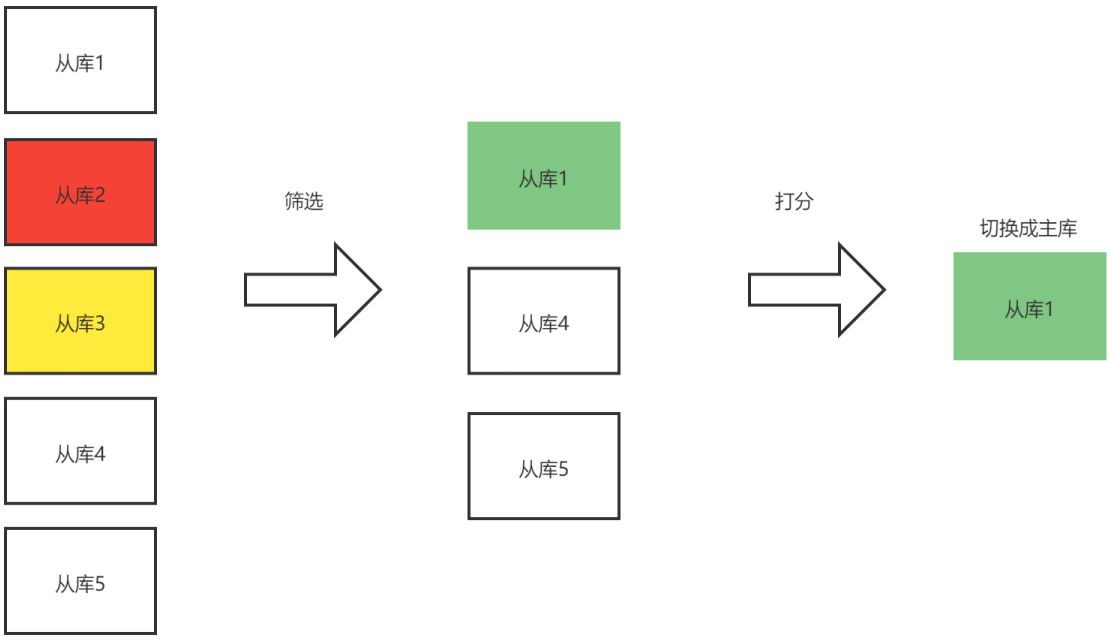
将一个从库6380，切换成主库，将6381，切换成6379的从库。

5.将原来主库6379再次启动，6379切换成6380的从库

9.4 新主库的选定

筛选 + 打分机制，来实现新主库的选定

筛选



打分

三轮打分

第一轮 优先级

通过replica-priority配置项，给不同的从库设置优先级。可以将内存大，网络好，配置高的从库优先级设置更高。

第二轮 和原主库同步程度

选择和原主库repl_backlog_buffer中的位置最接近的，做为分数最高

第三轮 ID号小的从库得分高

每一个redis实例都有一个id。

9.5哨兵集群

9.5.1 简介

采用多个哨兵，组成一个集群，以少数服从多数的原则，来判断主库是否已客观下线。

s/2+1

如果集群中，有哨兵实例掉线，其他的哨兵还可以继续协作，来完成主从库监控和切换的工作。

9.5.2 部署

1.创建了一个目录 mysentinel

2.分别创建三个哨兵配置文件

sentinel26379.conf　sentinel26380.conf　sentinel26381.conf

配置如下

```
port 26379
sentinel monitor mymaster 127.0.0.1 6379 2
```

```
port 26380
sentinel monitor mymaster 127.0.0.1 6379 2
```

```
port 26381
sentinel monitor mymaster 127.0.0.1 6379 2
```

3.再次配置一主二从

4.启动三个redis实例，配置成一主二从，6379是主库

5.依次启动三个哨兵实例。主库宕机，发现主库下线后，选举新的从库做为主库

+sdown	进入主观下线状态
-sdown	退出主观下线状态
+odown	进入客出下线状态
-odown	退出客观下线状态
+switch-master	主库地址发生变化切换
+slave-reconf-inprog	从库配置了新主库，但尚未进行同步
+slave-reconf-done	从库配置了新主库，并且已经完成同步
+slave-reconf-sent	哨兵发送replicaof命令配置从库

9.5.3运行机制

基于pub/sub(发布/订阅)机制实现哨兵集群组成

基于info命令对哨兵监控从库

基于哨兵自身的pub/sub功能，实现了客户和哨兵之间的通知

subscribe +odown

subscribe 频道[频道...]

publish 频道 内容

down-after-milliseconds

10.redis分片集群

10.1简介

业务场景，需要存储50G的数据。对于内存和硬盘配置不足，选用两种方式

一种：纵向扩展：加内存，加硬盘，提高CPU。简单、直接。RDB存储效率要考虑。成本要考虑。

二种：横向扩展：加实例。

10.2配置

步骤1：创建一个mycluster目录，复制redis.conf文件

```
mkdir mycluster
cp /opt/redis-6.2.6/redis.conf  redis.conf
```

将后台启动打开

步骤2：创建一个redis6379.conf

```
include redis.conf
pidfile "/var/run/redis_6379.pid"
port 6379
dbfilename "dump6379.rdb"
#打开集群模式
cluster-enabled yes
#设定节点配置文件
cluster-config-file nodes-6379.conf
#设定节点失联时间，超过，会自动进行主从切换
cluster-node-timeout 15000
```

步骤3:根据6379的配置文件，再配置5个不同的端口

```
:%S/原内容/要替换的内容
:%s/6379/6380
```

步骤4:修改redis.conf配置文件，将bind ip地址加入

```
打开bind 127.0.0.1
加入 192.168.44.4
```

步骤5 启动六个服务，保证六个服务启动成功

步骤6 将六个服务合成一个集群

```
redis-cli --cluster create --cluster-replicas 1 192.168.44.4:6379
192.168.44.4:6380 192.168.44.4:6381 192.168.44.4:6579 192.168.44.4:6580
192.168.44.4:6581
```

输入yes去接受配置

10.3 Hash Slot

10.3.1 简介

在使用redis cluster方案中，一个分片集群有16384个哈希槽。

根据键值对的key，按照CRC16算法计算一个16bit的值。再用这个值对16384取模运算，得到的数代表对应编号的hash slot

10.3.2部署方案

cluster create命令创建集群时，redis会自动把这些hash slot平均分布在集群实例上。如果集群中有N个实例(主库)，每个实例上分配到的hash slot就是16384/N

使用cluster addslos 手工分配哈希槽。

10.4 集群中的数据操作

步骤1：使用集群方式启动redis的命令行,加 -c参数

```
redis-cli -c -p 6379
```

步骤2：向redis中设置一个键值对，key会经过运算后，得到相应的hash slot进行存储

```
set k1 v1
set k2 v2
get k1
get k2
```

步骤3：测试向集群中加入多个key-value，由于在不同的hash slot，此时会报错

```
mset k3 v3 k4 v4 k5 v5
```

要通过{}来定义组的概念，使用key中{} 内相同内容的键值对放在一个slot中

```
mset name{user:001} zhangsan age{user:001} 18 gender{user:001} man
```

步骤4:获取key中的值

```
get k1
get name{user:001}
```

10.5 常用命令

- cluster nodes：显示集群节点的配置信息
- cluster keyslot < key>：获取key的哈希槽
- cluster countkeysinslot < slot>：返回当前哈希槽中key的数量(仅查询当前redis实例)
- cluster getkeysinslot < slot>< count>：返回当前槽中指定的count数量的key

10.6 故障演示

步骤1：将6379宕机，以集群方式登录6380

```
127.0.0.1:6379>shutdown
     exit
redis-cli -c -p 6380
```

步骤2:使用cluster nodes查看节点状态，6379的从机6581变成了主机

步骤3: 把6379再次启动，启动后，6379变成了6581的从机

如果有一段hash slot的主从节点都宕机，redis是否继续工作?

通过下面配置，默认是yes，如果主从都挂掉，整个集群就都挂掉

如果是no，就表示该hash slot数据全部都不能使用，也无法存储

```
cluster-require-full-coverage yes
```

11.亿级访问量数据处理

11.1 场景描述

- 手机APP用户登录信息，一天用户登录ID或设备ID
- 电商或者美团平台，一个商品对应的评论
- 文章对应的评论
- APP上有打卡信息
- 网站上访问量统计
- 统计新增用户第二天还留存
- 商品评论的排序
- 月活统计
- 统计独立访客(Unique Vistitor UV)量

11.2 集合的统计模式

四种统计模式：聚合统计、排序统计、二值状态统计、基数统计

11.2.1 聚合统计

多个集合的交集、差集、并集

set集合，来存储所有登录系统的用户 user:id

set集合，来存储当日新增用户信息 user:id :20211222

假设系统是2021年12月22日上线，统计当天用户

sadd user:id :20211222 1001 1002 1003 1004 1005

统计总用户量

sunionstore user:id user:id user:id :20211222

第2天12月23日上线用户

sadd user:id :20211223 1001 1003 1006 1007

统计当日新增用户

sdiffstore user:new user:id :20211223 user:id

统计第一天登录，第二天还在的用户

sinterstore user:save user:id :20211222 user:id :20211223

统计第一天登录，第二天流失的用户

sdiffstore user:rem user:id : 20211222 user:id :20211223

11.2.2排序统计

List、Set 、Hash 、ZSet四种集合中，List和Zset是属于有序的集合

一种使用List，通过lpush加入

一种使用Zset，按分数权重处理

11.2.3 二值状态统计

统计疫苗接种人数(没有接种0 接种1)、打卡(没有打卡0 打卡1)、签到。

bit位　1byte=8bit

redis提供一种扩展数据类型 bitmap。

可以把bitmap看到是一个bit数组

常用命令：

setbit

getbit

bitcount

统计一下，2022年1月份的一个上班打卡情况

```
setbit user:sign:202201 0 1
setbit user:sign:202201 2 1
setbit user:sign:202201 3 1

getbit user:sign:202201 0    //1
getbit user:sign:202201 1    //0

bitcount user:sign:202201    //4
```

统计1亿个用户，10天签到情况

bitop

操作

setbit user:sign:1222 0 1

.....

bitop and signmap user:sign:1222 user:sign:1223 user:sign:1224

11.2.4 基数统计

统计一个集合中不重复的元素个数，例如统计网页的UV

第一种，使用set或者hash来完成统计

sadd page1:uv u1001 u1002 u1003

scard page1:uv

存在的问题：如果数据量非常大，且页面多，访问人数非常多，造成内存紧张

第二种，Redis提供了HyperLogLog（hll）

HyperLogLog是用于统计基数的一种数据集合类型。优点在于当集合元素非常多，使用hll所需要的空间是固定且很小，使用12kb内存，可以存储2^64个元素的基数。缺点在于统计规则是基于概率完成的。会有0.81%左右的误差。如果统计1000万次，实际上可以是1100万 或900万人。

命令

pfadd page1:uv u1001 u1002 u1003

pfcount page1:uv

pfadd page2:uv u1001 u1004

pfmerge page:uv page1:uv page2:uv

pfcount page:uv

11.2.5小结

数据类型	聚合统计	排序统计	二值状态统计	基数统计
set	支持差集、交集、并集	不支持	不支持	支持精确统计，数据量大时占用内存较大
zset	支持差集、交集、并集	支持	不支持	支持精确统计，数据量大时占用内存较大
hash	不支持	不支持	不支持	支持精确统计，数据量大时占用内存较大
list	不支持	支持	不支持	不支持
bitmap	与、或、异或运算	不支持	支持	支持精确统计，数据量大时占用内存较大
hyperloglog	不支持	不支持	不支持	支持，采用概率算法，大数据量时，节省内存，但不精确

12.Geospatial

12.1简介

基于位置信息服务(Location-Based Service,LBS)的应用。Redis3.2版本后增加了对GEO类型的支持。主要来维护元素的经纬度。redis基于这种类型，提供了经纬度设置、查询、范围查询、距离查询、经纬度hash等一些相关操作

12.2GEO底层结构

1.约车系统，针对每一辆车，有一个唯一编号,车辆有行驶的经纬度

2.呼叫车辆，会暴露用户的经纬度，根据经纬度进行范围查找，进行匹配

3.把附近车辆找到后，车辆信息获取，将信息反馈给用户

第一种方式，可以使用hash来存储，但hash没有排序功能

第二种方式，geo底层结束 zset 来实现。需要将经纬度放在一起，生成一个权重的分数，按这个分数进行排序

GEOHash编码

编码的过程：

经度-180,180之间，按给定位数做N次二分区操作。

例如，N=5，做5次二分区操作，例如，坐标116.40

对经度进行编码

次数	最小	中间	最大	区间	编码
1	-180	0	180	[0,180]	1
2	0	90	180	[90,180]	1
3	90	135	180	[90,135]	0
4	90	112.5	135	[112.5,135]	1
5	112.5	123.75	135	[112.5,123.75]	0

11010

对纬度进行编码　39.96

次数	最小	中间	最大	区间	编码
1	-90	0	90	[0,90]	1
2	0	45	90	[0,45]	0
3	0	22.5	45	[22.5,45]	1
4	22.5	33.75	45	[33.75,45]	1
5	33.75	39.375	45	[39.375,45]	1

10111

经度:11010

纬度:10111

最后合成的编码：1110011101

12.3 GEO操作指令

geoadd < key> < longitude>< latitude>< member> [longitude latitude member...]：添加地理位置(经度 纬度 名称)

geopos < key>< member>[member...]：获取指定的位置坐标值

geodist < key>< member1>< member2>：获取两个位置之间的直线距离。单位:m 米 km 千米 ft英尺 mi英里

georadius < key>< longitude> < latitue> radius [m | km| fm| mi]，以经定的经纬度做为中心，找出给定半径内的位置

12.4 查找附近的人案例

geoadd nearby 116.511023 39.945711 person1 116.508257 39.946735 person2 116.513395 39.948035 person3 116.51415 39.945131 person4 116.508724 39.943194 person5 116.511526 39.943775 person6 116.509802 39.94419 person7 116.512317 39.946928 person8 116.505166 39.946265 person9 116.506316 39.946375 person10

georadius nearby 116.506316 39.946375 500 m desc count 10

13.redis事务操作

13.1事务简介

原子性(Atomicity)

一致性(Consistency)

隔离性(isolation)

持久性(durabiliby)

ACID

13.2 Redis事务

提供了multi、exec命令来完成

第一步，客户端使用multi命令显式地开启事务

第二步，客户端把事务中要执行的指令发送给服务器端，例如set 、get 、lpush，这些指令不会立即执行，进入一个队列中

第三步，客户端向服务器发送一个命令 exec，来完成事务提交。当服务器端收到这个指令后，实际去执行上一步中的命令队列.

```
multi
set k1 v1
set k2 v2
set k3 v3
get k1
exec


multi
set k4 v4
set k5 v5
discard  //取消
```

13.3 Redis的事务处理机制

13.3.1 原子性

第一种情况，在执行exec指令前，客户端发送操作命令有误,redis会报错并记录这个错误。此时，还可以继续发送命令操作，在执行exec命令之后，redis拒绝执行所有提交的指令，返回事务失败的结果。（保证了原子性）

```
multi
set k1 v1
get k1 v1
set k2 v2
exec
整个队列失败
```

第二种情况，向服务器发送指令，其中有指令和操作的数据类型不匹配，放入队列时并没有报错。使用lpop指令操作失败，但get指令成功了。（不能保证原子性）

```
multi
lpop k1  //失败
get k1   //成功
exec
```

第三种情况，在执行事务的exec指令时，redis实例发生了故障,导致事务执行失败

如果redis开启了aof日志，可能会有一部分指令被记录到AOF日志中，需要使用redis-check-aof 去检查aof文件，将未完成事务操作从aof清除，从而保证原子性

13.3.2 一致性

- 第一种情况，指令进入队列时就报错，整个事务全部被放弃执行，可以保证数据的一致性。
- 第二种情况，进入队列时没有报错，实际执行时报错，有错误的指令不去执行，正确的指令可以正常执行，可以保证数据的一致性
- 第三种情况，exec指令时redis实例发生故障，根据RDB和AOF情况来做判断
    - 如果没有开启rdb和aof，数据在重启后没有，一致的
    - 如果使用了rdb方式，rdb不会在事务执行的时候去保存数据，数据库也是一致的
    - 使用aof日志，如果事务队列操作记录没有进入aof，可以保证一致性。如果已加入了一部分，使用redis-check-aof清除事务中已完成的操作，保证事务的一致性

13.3.3 隔离性

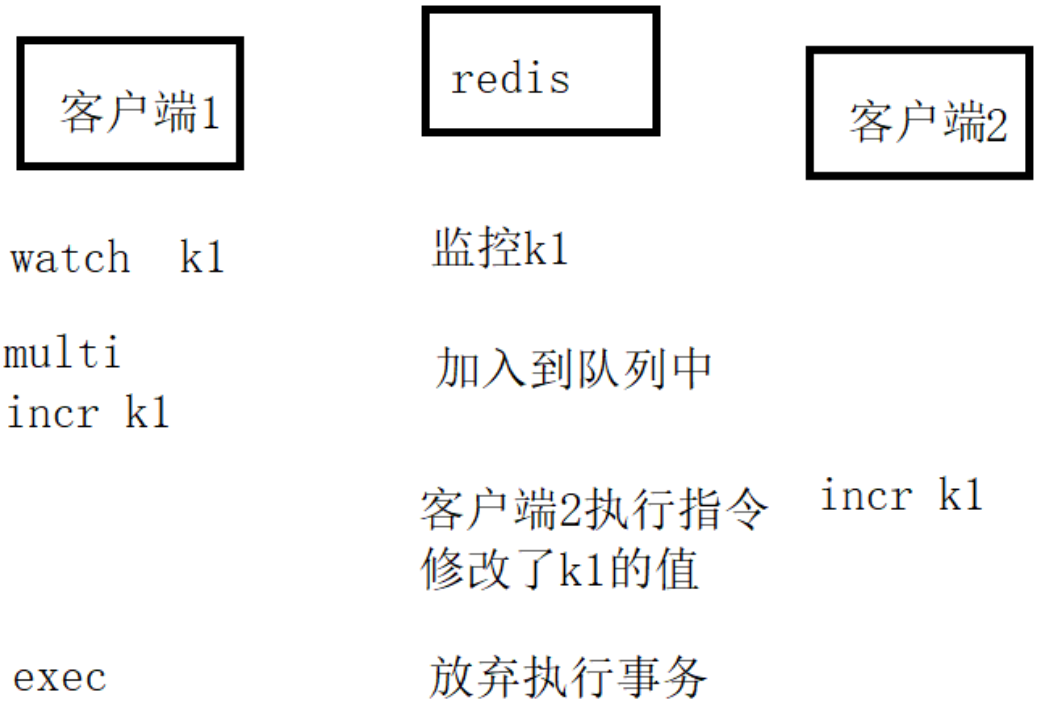
提交exec指令去执行事务，分成exec之前和exec之后两种情况

并发操作在exec指令前，要实现隔离性的保证 ，需要使用watch机制，否则不能保证隔离性

在事务执行前，相当于有一个监控器，在监控key是否已经被修改过了，如果已修改，则放弃事务执行，避免了事务的隔离性被破坏。如果客户再次执行，此时，没有其他客户端去修改数据，则执行成功。

悲观锁：synchronized

乐观锁：Atomic原子操作



使用unwatch取消watch命令对所有key的监控。

13.3.4 持久性

redis内存数据库，取决于持久化配置模式

不开启rdb和aof，只当作缓存使用，是不能保证持久性

使用rdb，如果在一个事务执行后，下一次的rdb快照还未执行前，redis实例发生故障了，不能保证持久性

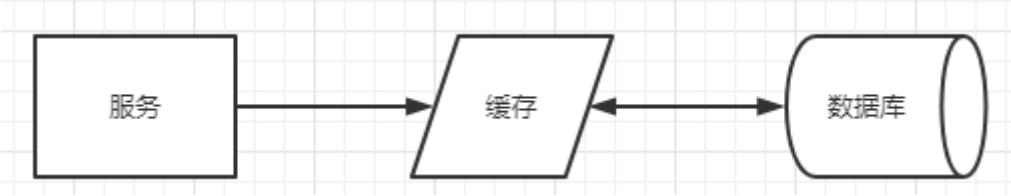
使用aof， 配置选项 everysec、always、no，也不能保证持久性

不管redis采用什么配置模式，都不能保证事务的持久性

14.redis缓存

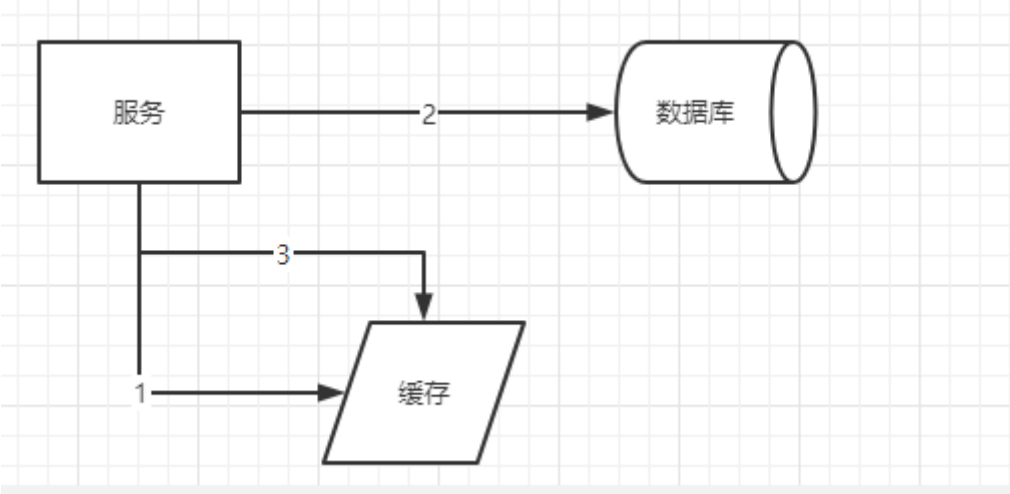
14.1简介

穿透型缓存：



缓存与后端数据交互在一起，对服务端的调用隐藏细节。如果从缓存中可以读到数据，就直接返回，如果读不到，就到数据库中去读取，从数据库中读到数据，也是先更新缓存，再返回给服务端。向数据库中写入数据，也是先写入缓存中，再同步给数据库

旁路型缓存：



1. 服务先到缓存中读取数据，如果数据存在，就直接返回
2. 如果缓存中没有数据，就到数据库中去读取
3. 服务再将从数据库中读到的数据同步给缓存

redis是旁路型缓存

14.2 缓存的特征

1.效率高

2.容量小

14.3 redis缓存处理的两种情况

缓存命中：redis中有相应的数据，直接从redis中读取，性能很高

缓存缺失：redis中没有相应的数据，从后端关系型数据库中读取数据，性能很低。如果发生缓存缺失，为了后续程序请求中可以从缓存中读取数据，要将缺失的数据写入redis,也称作缓存更新。

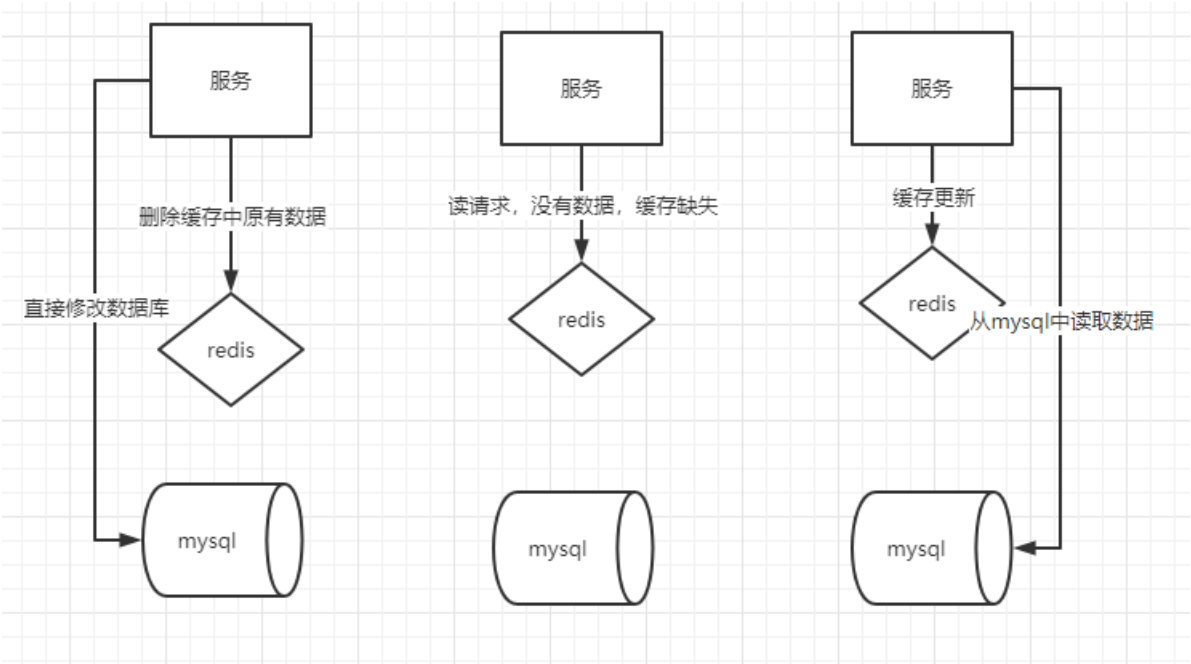
```
String cache_key="user1001";
String cache_value=redis.get(cache_key);//想要从缓存中读取数据

if (cache_value!=null){
    //做相关业务逻辑
}else{
    cache_value=mysql.getUserById(cache_key);//从关系型数据库中去读取数据
    redis.set(cache_key,cache_value);//缓存更新处理
}
```

redis不适用于无法获取源码的应用。

14.4 缓存的类型

14.4.1 只读缓存

只用读取数据的缓存。如果有写入数据的请求，直接发到后端的mysql或oracle数据库，在数据库中完成增删除改。对于删除和修改的数据来说，redis中可能会有旧的数据，需要将旧的数据删除，下一次读取时，redis缓存缺失，那么就从数据库中读数据，并更新到redis缓存中。



缓存图片、视频、手机的通讯记录、银行的以往帐单。

14.4.2 读写缓存

读定缓存，不只完成对数据读取任务，数据的增加、删除、修改操作，也是在redis缓存中完成，由于redis内存数据库效率很高，所以可以快速响应给服务端调用。

redis内存数据，在redis实例出现问题时，导致数据丢失。

同步直写：优先保证数据可靠

异步写回：执行效率高

对写请求操作进行高效处理，选择读写缓存

如果写操作很少，需要提升读取效率，选择只读缓存

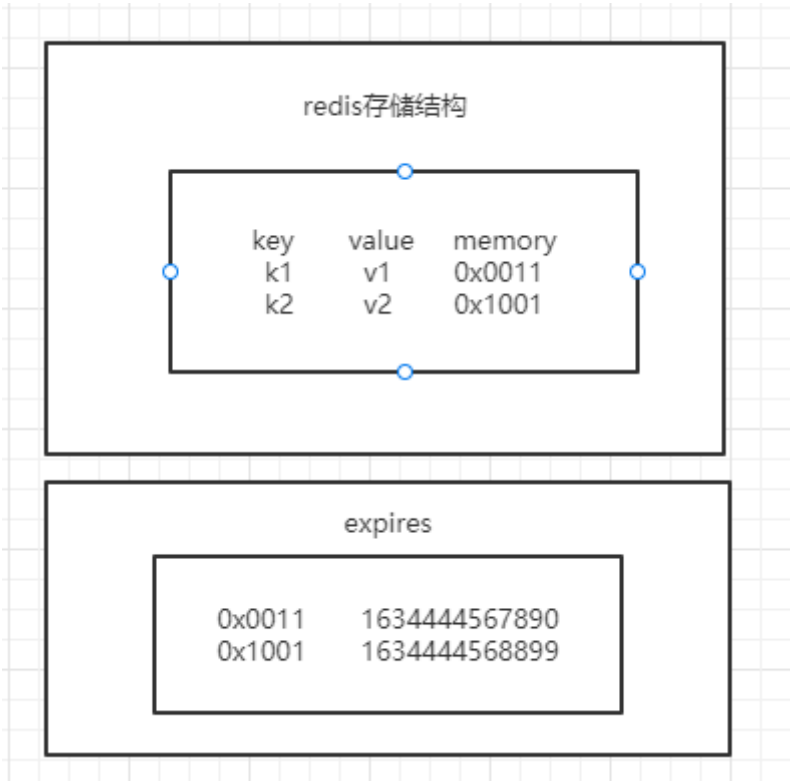
14.5 缓存数据的删除和替换

14.5.1 过期数据

可以使用ttl查看key的状态。已过期的数据，redis并未马上删除。优先去执行读写数据操作，删除操作延后执行。

14.5.2 删除策略

redis中每一个value对应一个内存地址，在expires，一个内存地址，对应一个时间截，如果达到指定时间，就完成删除处理



三种删除策略

定时删除：创建一个定时器，当key设置过期时间已到达，删除key,同时expires中也删除

优点：节约内存

缺点：对于cpu实时处理压力影响，对redis执行的效率有影响

惰性删除：数据到达过期时间，先不做删除，直到下次访问该数据时，再做删除

执行流程：在get数据时，先执行redis中一个内部函数 expireIfNeeded()，如果没有过期，就返回，如果已过期，就删除，返回-2

优点：节约CPU资源

缺点：内存占用过大。

定期删除

redis启动服务时，读取server.hz的值，默认为10，可以通过info server指令查看

每秒钟执行server.hz次定时轮询，调用serverCron()函数，函数中又执行databasesCron()，对16数据库进行轮询，执行了activeExpireCycle()，检测其中元素的过期情况。每次轮询都执行250ms/server.hz时长。随机从对应的库中抽取20个(默认)key进行检测

如果key已过期，则删除key

如果一轮中删除的key数量>w*25%，则再次循环刚才的过程

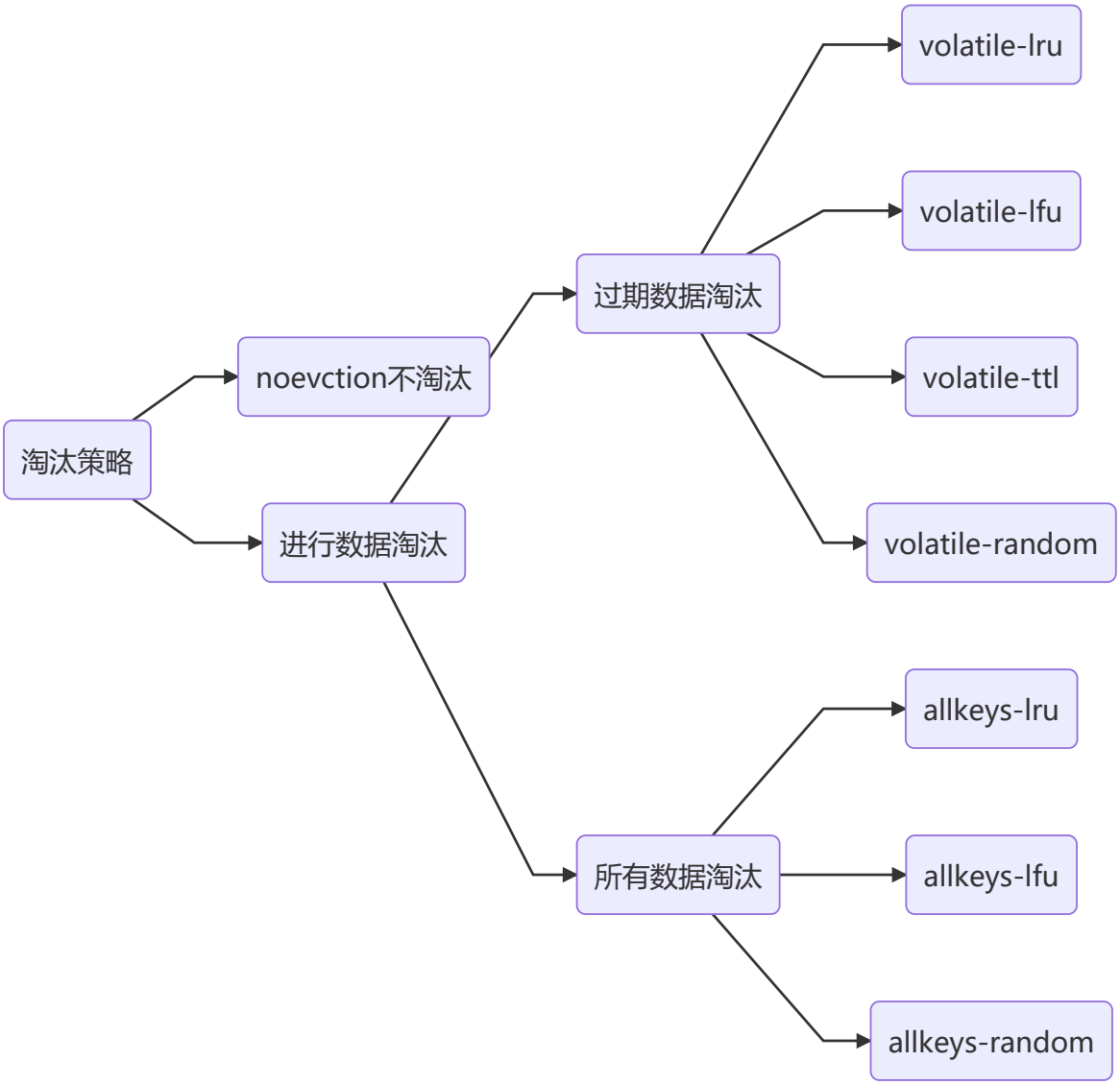
如果一轮中删除的key数量<=w*25%，则开始检查下一个库

redis中使用惰性删除和定期删除

14.5.3 逐出算法

通过 配置文件 maxmemory < bytes>来设置最大缓存容量。一般情况，建议设置为总数据的15%到30%，在实际生产环境下，可以设置50%。如果不设置，默认全部使用

redis缓存淘汰策略



在redis默认情况下，不进行数据淘汰noevction,一旦缓存被写满了，再有写请求，redis直接返回错误。

过期数据淘汰策略，先限定了，数据都是在过期范围。

- valotile-ttl：在进行筛选时，根据过期时间先后顺序进行一个删除，越早过期的越先被删除
- valotile-random：在设置了过期时间的键值对中，进行随机删除
- valotile-lru：会使用LRU算法筛选设置了过期的键值对
- valotile-lfu：会使用LFU算法筛选设置了过期的键值对

所有数据淘汰策略：

- allkeys-random：从所有键值对中随机筛选并删除

- allkeys-lru：从所有键值对中采用LRU算法进行筛选删除
- allkeys-lfu：从所有键值对中采用LFU算法进行筛选删除

LRU算法

算法Least Recently Used，最近最少使用原则，最近不用的数据会被筛选出来，最近频繁使用的数据会保留

lru算法，需要使用链表来管理所有缓存数据，带来内存开销。有数据被访问时，需要执行链表数据的移动，会降低redis性能。

记录数据最后一次访问的时间截，第一次会随机选出N个数据,作为一个候选集合，作一个排序，再把lru最小的数据进行淘汰

N的配置

maxmemory-samples 5

LFU算法

算法Least Frequently Used，最不常用原则。根据历史访问频率来淘汰数据。

每个数据块都有一个引用计数,按引用计数来排序。如果引用计数相同，按照时间排序

- 新加入的数据放在队尾，引用计为1
- 当数据被访问，引用计数增加，队列重排
- 当需要淘汰数据时，将队列尾部的数据块删除

逐出算法选择

maxmemory-policy noeviction

优先使用allkeys-lru策略。

如果业务数据访问频率差别不大，可以建议使用allkeys-random。

首推的新闻、置顶视频，不设置过期时间，可以建议使得volatile-lru。

14.6缓存异常

四个方面：缓存中数据和数据库不一致，缓存雪崩，缓存击穿和缓存穿透

14.6.1数据不一致：

一致性包括两种情况

- 缓存中有数据，需要和数据库值相同
- 缓存中没有数据，数据库中的数据是最新值

如果不符合以上两种情况，则出现数据不一致的问题。

读写缓存

- 同步直写
- 异步写回

只读缓存

1、新增数据

数据直接写到数据库中，缓存不做操作。满足一致性两种情况的第2种。

2、删改数据

先删除缓存，后更新数据库。可能会导致，缓存删除成功，数据库更新失败。业务逻辑去访问数据时，缓存中查不到数据，缓存缺失，到数据库中查询，所以只拿到旧的数据。

如果新更新数据库，再删除缓存。可能会导致，数据库更新成功，缓存删除失败。数据库中的数据是新的值，缓存中存储的是旧值。再读取时，先从缓存中读取，读取到了旧值。

解决数据不一致的方案

重试机制：把删除的缓存值或要更新数据库值先存储到消息队列中(kafka消息队列)。

多线程访问的情况

1.先删除缓存，再更新数据库。

假设T1线程先删除缓存，再执行更新数据库。还未更新成功时，T2线程进行读取，发现缓存中没有数据，到数据库中读取，会读取到旧的数据。如果T2还将旧数据更新到缓存中，那T1线程再进行读取，也读到的旧值。

让T1线程先执行休眠一段时间。T1线程在休眠时间，让T2线程执行结束，会将数据重新写入缓存。T1线程再做一次缓存删除操作。"延迟双删"。

```
redis.delCache()

db.update()

Thread.sleep(2000)

redis.delCache()
```

2.先更新数据库值，再去删除缓存

假设T1线程先删除或更新数据库中的值，还没来得及删除缓存时，T2线程就开始读取数据。T2会先从缓存中读取，缓存命中，T2拿到的就是旧的数据。直到T1将缓存中数据删除，其他线程再次读取，可以拿到新值.

并发操作	**顺序**	**可能出现问题**	**问题描述**	**解决方案**
没有	先删除缓存，后再新数据库	缓存删除成功，数据库更新失败	从数据库读到旧值	重试(通过消息队列)
有	先删除缓存，后更新数据库	缓存删除，未更新数据库，其他线程并发访问	并发线程从数据库读到旧值，并更新了缓存，其他线程都从缓存中读到旧值	延迟双删
没有	先更新数据库，后删除缓存	数据库更新成功，缓存删除失败	从缓存中读到旧值	重试(通过消息队列)
有	**先更新数据库，后删除缓存**	数据库更新成功，未删除缓存，其他线程并发访问	并发线程从缓存中读到旧值	会有数据不一致情况短暂存在

14.6.2 缓存雪崩

大量的应用请求无法在redis中完成处理。缓存中读取不到数据，直接进入到数据库服务器。数据库压力激增，数据库崩溃，请求堆积在redis，导致redis服务器崩溃，导致redis集群崩溃，应用服务器崩溃，称为雪崩

原因1：缓存中有大量数据同时过期

解决方案：

1.页面静态化处理数据：对于不经常更换的数据，生成静态页

2.避免大量数据同时过期:为商品过期时间追加一个随机数，在一个较小的范围内(1~3分钟)。

3.构建多级缓存架构：redis缓存+nginx缓存+ehcache缓存

4.延长或取消热度超高的数据过期时间

5.服务降级

不同的数据采取不同的处理方式。

原因2：redis实例故障

解决方案：

1.服务熔断或限流处理

提前预防

灾难预警：监控redis服务器性能指标，包括数据库服务器性能指标，CPU、内存、平均响应时间、线程数等

集群：有节点出一故障，主从切换。

14.6.2 缓存击穿

对某个访问频繁热点数据的请求。主要发生在热点数据失效

解决方案：

预先设定：电商双11,商铺设定几款是主打商品，延长过期时间

实时监控：监控访问量，避免访问量激增

定时任务：启动任务调度器，后台刷新数据有效期

分布式锁：可防止缓存击穿，但会有性能问题

14.6.3 缓存穿透

要访问的数据在redis中不存在，在数据库中也不存在。

原因：

1.业务层误操作

2.恶意攻击

解决方案：

1.缓存空值或缺省值

2.使用布隆过滤器，快速判断数据是否存在

3.在请求入口前端进行请求检测

4.实时监控

5.key加密

15.使用jedis操作

15.1 连接redis

步骤1.创建一个maven工程

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.example</groupId>
    <artifactId>redis_demo</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>11</maven.compiler.source>
        <maven.compiler.target>11</maven.compiler.target>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>1.18.22</version>
        </dependency>
        <dependency>
            <groupId>ch.qos.logback</groupId>
            <artifactId>logback-classic</artifactId>
            <version>1.2.7</version>
        </dependency>
        <dependency>
            <groupId>redis.clients</groupId>
            <artifactId>jedis</artifactId>
            <version>4.0.1</version>
        </dependency>
    </dependencies>
</project>
```

步骤2.配置logback用到的xml文件 logback.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration
        xmlns="http://ch.qos.logback/xml/ns/logback"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://ch.qos.logback/xml/ns/logback logback.xsd">
    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%date{HH:mm:ss.SSS} %c [%t] - %m%n</pattern>
        </encoder>
    </appender>
    <logger name="c" level="debug" additivity="false">
        <appender-ref ref="STDOUT"/>
    </logger>
```

```
    <root level="ERROR">
        <appender-ref ref="STDOUT"/>
    </root>
</configuration>
```

步骤3：修改redis.conf的配置文件 ，加入当前虚拟机的ip地址

bind 192.168.44.4 127.0.0.1 -::1

步骤4：修改redis.conf的配置文件，将文件保护模式修改，原有默认值是yes，改成no

protected-mode no

步骤5：查看一下linux防火墙状态

systemctl status firewalld

默认是打开的 绿色active(running)

步骤6：关闭防火墙

systemctl stop firewalld

关闭之后再去查看防火墙状态

inactive (dead)

步骤7：测试连接

```
package com.xzit.test;

import lombok.extern.slf4j.Slf4j;
import redis.clients.jedis.Jedis;

@Slf4j(topic = "c.TestConnection")
public class TestConnection {
    public static void main(String[] args) {
        Jedis jedis=new Jedis("192.168.44.4",6379);
        String p = jedis.ping();
        log.debug(p);
    }
}
```

反馈结果

```
23:33:01.588 c.TestConnection [main] - PONG
```

15.2 操作String类型

```
package com.xzit.test;

import lombok.extern.slf4j.Slf4j;
import redis.clients.jedis.Jedis;

import java.util.List;

@Slf4j(topic = "c.TestString")
public class TestString {
    public static void main(String[] args) {
```

```
        Jedis jedis=new Jedis("192.168.44.4",6379);
        //set(jedis,"k2","v2");//设置
        //log.debug(get(jedis,"k2"));//获取
        //m_set(jedis,"k3","v3","k4","v4");
       // m_get(jedis,"k2","k3");
        //set_ex(jedis,"k5",30,"v5");
        ttl(jedis,"k5");
    }
    public static void set(Jedis jedis, String key, String value){
        jedis.set(key,value);
    }
    public static String get(Jedis jedis, String key){
        return jedis.get(key);
    }
    public static void m_set(Jedis jedis,String...keysvalues){
        jedis.mset(keysvalues);
    }
    public static void m_get(Jedis jedis,String...keys){
        List<String> list = jedis.mget(keys);
        list.forEach(System.out::println);
       /* for (String s : list) {
            System.out.println(s);
        }*/
    }
    public static void set_ex(Jedis jedis,String key,long seconds,String value){
        jedis.setex(key,seconds,value);
    }
    public static void ttl(Jedis jedis,String key){
        long ttl = jedis.ttl(key);
        log.debug("{}",ttl);
    }
}
```

15.3 操作List类型

```
package com.xzit.test;

import lombok.extern.slf4j.Slf4j;
import redis.clients.jedis.Jedis;

import java.util.List;

@Slf4j(topic="c.TestList")
public class TestList {
    public static void main(String[] args) {
        Jedis jedis=new Jedis("192.168.44.4",6379);
        //l_push(jedis,"k1","aaa","bbb","ccc");
        //l_range(jedis,"k1",0,-1);
        //l_pop(jedis,"k1",2);
        l_rem(jedis,"k1",2,"bbb");
    }
    public static void l_push(Jedis jedis,String key,String...values){
        jedis.lpush(key,values);
    }
    public static void l_range(Jedis jedis,String key,long start,long stop){
        List<String> list = jedis.lrange(key, start, stop);
```

```
        list.forEach(c->log.debug(c));
    }
    public static void l_pop(Jedis jedis,String key,int count){
        List<String> lpop = jedis.lpop(key, count);
        lpop.forEach(c->log.debug(c));
    }
    public static void l_rem(Jedis jedis,String key,long count,String value){
        long lrem = jedis.lrem(key, count, value);
        log.debug("成功删除{}",lrem);
    }
}
```

15.4 操作Hash类型

```
package com.xzit.test;

import lombok.extern.slf4j.Slf4j;
import redis.clients.jedis.Jedis;

import java.util.HashMap;
import java.util.Map;
import java.util.Set;

@Slf4j(topic = "c.TestHash")
public class TestHash {
    public static void main(String[] args) {
        Jedis jedis=new Jedis("192.168.44.4",6379);
        //h_set(jedis,"user:1001","name","张三");
       // h_set(jedis,"user:1001","age","20");
       // h_get(jedis,"user:1001","name");
        /*Map<String,String> map=new HashMap<>();
        map.put("name","lisi");
        map.put("age","20");
        map.put("gender","man");
        h_set(jedis,"user:1002",map);*/
       // h_getAll(jedis,"user:1002");
        h_keys(jedis,"user:1002");
    }
    public static void h_set(Jedis jedis,String key,String field,String value){
        jedis.hset(key,field,value);
    }
    public static void h_set(Jedis jedis, String key, Map<String,String> value){
        jedis.hset(key,value);
    }
    public static void h_get(Jedis jedis,String key,String field){
        String value = jedis.hget(key, field);
        log.debug(value);
    }
    public static void h_getAll(Jedis jedis,String key){
        Map<String, String> map = jedis.hgetAll(key);
        map.forEach((k,v)->log.debug(k+"\t"+v));
       /* for (String key1:map.keySet()){
            log.debug(key1+"\t"+map.get(key1));
        }*/
    }
    public static void h_keys(Jedis jedis,String key){
```

```
        Set<String> fields = jedis.hkeys(key);
        fields.forEach(System.out::println);

    }
}
```

15.5 操作Set类型

```
package com.xzit.test;

import lombok.extern.slf4j.Slf4j;
import redis.clients.jedis.Jedis;

import java.util.Set;

@Slf4j(topic = "c.TestSet")
public class TestSet {
    public static void main(String[] args) {
        Jedis jedis=new Jedis("192.168.44.4",6379);
        //s_add(jedis,"k1","aaa","bbb","ccc","ddd");
       // s_add(jedis,"k2","bbb","ccc","ddd","eee");
        //s_members(jedis,"k1");
        //s_ismember(jedis,"k1","aaaa");
        //scard(jedis,"k1");
        //s_rem(jedis,"k1","aaa","bbb");
        s_diff(jedis,"k2","k1");
    }
    public static void s_add(Jedis jedis,String key,String...members){
        jedis.sadd(key,members);
    }
    public static void s_members(Jedis jedis,String key){
        Set<String> set = jedis.smembers(key);
        set.forEach(System.out::println);
    }
    public static void s_ismember(Jedis jedis,String key,String member){
        boolean b = jedis.sismember(key, member);
        log.debug("{}",b);
    }
    public static void scard(Jedis jedis,String key){
        long scard = jedis.scard(key);
        log.debug("{}",scard);
    }
    public static void s_rem(Jedis jedis,String key,String...members){
        long srem = jedis.srem(key, members);
        log.debug("{}",srem);
    }
    public static void s_diff(Jedis jedis,String...keys){
        Set<String> sdiff = jedis.sdiff(keys);
        sdiff.forEach(System.out::println);
    }
}
```

15.6 操作zset类型

```
package com.xzit.test;

import lombok.extern.slf4j.Slf4j;
import redis.clients.jedis.Jedis;

import java.util.HashMap;
import java.util.List;
import java.util.Map;

@Slf4j(topic = "c.TestZset")
public class TestZset {
    public static void main(String[] args) {
        Jedis jedis=new Jedis("192.168.44.4",6379);
        //z_add(jedis,"k1",100,"元旦假期");
        //z_add(jedis,"k1",110,"清明假期");
        //z_add(jedis,"k1",120,"五一假期");
       // z_range(jedis,"k1",0,-1);
       /* Map<String,Double> map=new HashMap<>();
        map.put("元旦假期",100.0);
        map.put("清明假期",110.0);
        map.put("五五假期",120.0);
        z_add(jedis,"k2",map);*/
        //z_range(jedis,"k2",0,-1);
        //z_rev_range(jedis,"k2",0,-1);
        //z_rank(jedis,"k1","清明假期");
        z_card(jedis,"k1");
    }
    public static void z_add(Jedis jedis,String key,double score,String member){
        jedis.zadd(key,score,member);
    }
    public static void z_add(Jedis jedis, String key, Map<String,Double>
members){
        jedis.zadd(key,members);
    }
    public static void z_range(Jedis jedis,String key,long start,long stop){
        List<String> list = jedis.zrange(key, start, stop);
        list.forEach(System.out::println);
    }
    public static void z_rev_range(Jedis jedis,String key,long start,long stop){
        List<String> list = jedis.zrevrange(key, start, stop);
        list.forEach(System.out::println);
    }
    public static void z_rank(Jedis jedis,String key,String member){
        Long index = jedis.zrank(key, member);
        log.debug("{}",index);
    }
    public static void z_card(Jedis jedis,String key){
        long zcard = jedis.zcard(key);
        log.debug("{}",zcard);
    }

}
```

15.7 操作geo类型

```
package com.xzit.test;
```

```
import lombok.extern.slf4j.Slf4j;
import redis.clients.jedis.GeoCoordinate;
import redis.clients.jedis.Jedis;
import redis.clients.jedis.args.GeoUnit;
import redis.clients.jedis.resps.GeoRadiusResponse;

import java.util.List;

@Slf4j(topic="c.TestGeo")
public class TestGeo {
    public static void main(String[] args) {
        Jedis jedis=new Jedis("192.168.44.4",6379);
        //geo_add(jedis,"nearby",116.511023,39.945711,"person1");
        //geo_add(jedis,"nearby",116.508257,39.946735,"person2");
        //geo_dist(jedis,"nearby","person1","person2");
        //geo_radius(jedis,"nearby",116.511023,39.945711,500);
        geo_pos(jedis,"nearby","person1","person2");
    }
    public static void geo_add(Jedis jedis,String key,double longitude,double
latitude,String member){
        jedis.geoadd(key,longitude,latitude,member);
    }
    public static void geo_dist(Jedis jedis,String key,String member1,String
member2){
        Double d = jedis.geodist(key, member1, member2);
        log.debug("{}",d);
    }
    public static void geo_pos(Jedis jedis,String key,String...members){
        List<GeoCoordinate> list = jedis.geopos(key, members);
        list.forEach(g->{
            log.debug(g.getLongitude()+","+g.getLatitude());
        });
    }
    public static void geo_radius(Jedis jedis,String key,double longitude,double
latitude,double radius){
        List<GeoRadiusResponse> list = jedis.georadius(key, longitude, latitude,
radius, GeoUnit.M);
        list.forEach(g->{
            log.debug(g.getMemberByString());
        });
    }
}
```

15.8 应用案例

需求：用户使用银行APP登录，要求动态发送手机验证码，验证码设定5分钟有效，1天内最多发送3次

```
package com.xzit.test;

import lombok.extern.slf4j.Slf4j;
import redis.clients.jedis.Jedis;

import java.text.DecimalFormat;
import java.util.Random;
```

```
//需求：用户使用银行APP登录，要求动态发送手机验证码，验证码设定5分钟有效，1天内最多发送3次
@Slf4j(topic="c.TestApp")
public class TestApp {
    public static void main(String[] args) {
        Jedis jedis=new Jedis("192.168.44.4",6379);
        //send(jedis,"13012345678");
        verify(jedis,"13012345678","067096");

    }
    public static String getCode(){
        return new DecimalFormat("000000").format(new
Random().nextInt(1000000));
    }
    public static void send(Jedis jedis,String phoneNumber){
        String count_key="v:"+phoneNumber+":count";//当前手机号已发送次数的key
        String code_key="v:"+phoneNumber+":code";//当前手机号已收到的验证码key
        String s=jedis.get(count_key);//获取手机号已发送的次数
        if (s==null){
            //如果没有获取，就表时该key不存在，第一次设置，有效时间为1天
            jedis.setex(count_key,60*60*24,"1");
            log.debug("发送成功");
        }else if(Integer.parseInt(s)<3){//如果不到3次，可以为用户发送，每发送一次，记数
增加1
            jedis.incr(count_key);
            log.debug("发送成功");
        }else{
            log.debug("今日转帐3次，24小时后再试");
            jedis.close();
            return;
        }
        jedis.setex(code_key,60*5,getCode());//发送验证码进行保存
        jedis.close();
    }
    public static void verify(Jedis jedis,String phoneNumber,String code){
        String code_key="v:"+phoneNumber+":code";
        String redis_code=jedis.get(code_key);
        log.debug(code.equals(redis_code)?"成功":"失败");
    }
    /*public static String getCode(){
        String v="0123456789";
        String code="";
        for (int i=1;i<=6;i++){
            code+=v.charAt((int)(Math.random()*v.length()));
        }
        return code;
    }*/
}
```