

Redis6

Redis6

一、为什么使用NoSQL?

- 1.1 NoSQL概述
- 1.2 NoSQL的使用场景
- 1.3 NoSQL不适用的场景
- 1.4 常见的NoSQL
 - 1、Memcache
 - 2、MongoDB
 - 3、Redis

二、Redis概述与安装

- 2.1 Redis概述
- 2.2 Redis安装
 - 2.2.1 安装步骤
- 2.3 启动Redis
 - 2.3.1 前台启动 (不推荐)
 - 2.3.2 后台启动 (推荐)
- 2.4 Redis 必知点

三、常用五大数据类型

- 3.1 键(Key)
- 3.2 String字符串
 - 3.2.1 常用命令
 - 3.2.2 数据结构
- 3.3 List列表
 - 3.3.1 常用命令
 - 3.3.2 数据结构
- 3.3 Set集合
 - 3.3.1 常用命令
 - 3.3.2 数据结构
- 3.4 Hash哈希
 - 3.4.1 常用命令
 - 3.4.2 数据结构
- 3.5 Zset有序集合
 - 3.5.1 常用命令
 - 3.5.2 数据结构
 - 3.5.2.1 跳跃表

四、Redis配置文件

- 4.1 Units单位
- 4.2 includes包含
- 4.3 网络相关配置
 - 4.3.1 bind
 - 4.3.2 protected-mode
 - 4.3.3 Port
 - 4.3.4 tcp-backlog
 - 4.3.5 timeout

4.3.6 tcp-keepalive

4.4 通用配置

4.4.1 daemonize

4.4.2 pidfile

4.4.3 logfile

4.4.4 loglevel

4.4.4 databases 16

4.5 安全配置

4.5.1 设置密码

4.6 限制配置

4.6.1 maxclients

4.6.2 maxmemory

4.6.3 maxmemory-policy

4.6.4 maxmemory-samples

五、Redis的发布与订阅

5.1 什么是发布与订阅

5.2 图片演示

5.3 发布订阅命令行实现

六、新数据类型

6.1 Bitmaps位图

6.2 HyperLogLog基数集

6.3 Geospatial地理空间

七、Jedis操作Redis

八、SpringBoot集成Redis

九、Redis事务、锁(秒杀案例)

9.x multi exec discard

9.x lua 脚步

十、Redis持久化

10.1 RDB

10.2 AOF

十一、主从复制

11.1 复制原理

11.2 常用三招

11.2.1 一主二仆

11.2.2 薪火相传

11.2.3 反客为主

11.3 哨兵模式

十二、集群

12.1 什么是集群?

12.2 模拟集群

12.2.1 删除持久化数据

12.2.2 制作六个实例

12.2.3 将六个节点合成一个集群

12.2.4 集群模式下登录客户端

12.2.5 redis cluster 如何分配节点

12.3 slots 插槽

12.4 在集群中录入值

12.6 查询集群中的值

- 12.7 故障修复
- 12.8 jedis集群开发
- 12.9 Redis集群好处
- 12.10 Redis集群的不足

十三、Redis应用解决问题

- 13.1 缓存穿透
 - 13.1.1 问题描述
 - 13.1.2 解决方案
- 13.2 缓存击穿
 - 13.2.1 问题概述
 - 13.2.2 解决方案
- 13.3 缓存雪崩
 - 13.3.1 问题描述
 - 13.3.2 解决方案
- 13.4 分布式锁
 - 13.4.1 问题描述
 - 13.4.2 使用redis实现分布式锁
 - 13.4.3 代码实现
 - 13.4.3.1 Springboot连接redis
 - 13.4.3.2 配置文件
 - 13.4.3.3 java代码
 - 13.4.3.4 ab(网关压力)测试
 - 13.4.4 优化之设置过期时间
 - 13.4.5 优化之UUID防误删
 - 13.4.6 优化之LUA脚本保证删除的原子性
 - 13.4.7 总结

十四、Redis6新功能

- 14.1 ACL
 - 14.1.1 什么是ACL
 - 14.1.2 命令
- 14.2 IO多线程
 - 14.2.1 原理架构
- 14.3 工具支持Cluster
- 14.4 Redis新功能持续关注

一、为什么使用NoSQL?

当今互联网时代，用户访问量大幅提升，出现了高并发高内存问题，同时产生了大量的用户数据，为了缓解CPU及

内存压力，NoSQL这种缓存数据库的作用就体现出来了，它使得数据完全在内存中，访问数据速度极快，数据结

构也较简单，同时它还解决了IO压力，以往的关系型数据库当存储的内容较大时，会选择"水平切分，垂直切分，

读写分离等操作"但是这破坏了一定的业务逻辑依此来换取性能，代价还是很大的。总而言之，NoSQL的出现是解

决性能方面的。

1.1 NoSQL概述

NoSQL (NoSQL = Not Only SQL)，意思“不仅仅是 SQL”，泛指非关系型的数据库。

NoSQL 不依赖业务逻辑方式存

储，而以简单的 **key-value** 模式存储。因此大大的增加了 数据库的扩展能力。

其特点：

- 不遵循SQL标准
- 不支持ACID (事务四大特性，不是说Redis不支持事务)
- 远超于SQL的性能

1.2 NoSQL的使用场景

- 存在数据高并发的读写
- 海量数据的读写
- 对数据可扩展性的

总之：数据量庞大存在高并发且对数据可扩展的选择NoSQL

1.3 NoSQL不适用的场景

- 需要事务的支持
- 处理较为复杂的关系

1.4 常见的NoSQL

1、Memcache



1. 很早出现的NoSql数据库
2. 数据都在内存中，一般不持久化
3. 支持简单的k-v模式，支持的数据类型较为单一
4. 一般作为缓存数据库辅助持久化的数据库

2、MongoDB



1. 高性能、开源、模式自由(schema free)的文档型数据库
2. 数据都在内存中，如果内存不足，把不常用的数据保存到硬盘
3. 虽然是 key-value 模式，但是对 value (尤其是 json) 提供了丰富的查询功能
4. 支持二进制数据及大型对象，
5. 可以根据数据的特点替代 RDBMS，成为独立的数据库。或者配合 RDBMS，存储特定的数据

3、Redis



1. 几乎覆盖了 Memcached 的绝大部分功能
2. 数据都在内存中，支持持久化，主要用作备份恢复
3. 除了支持简单的 key-value 模式，还支持多种数据结构的存储，比如 list、set、hash、zset 等。
4. 一般是作为缓存数据库辅助持久化的数据库

二、Redis概述与安装

2.1 Redis概述

Redis是一种基于内存存储和运行的，能快速响应的键值数据库产品。它的英文全称是 **Remote Dictionary Server**(远程字典服务)

- **Redis是一个开源的k-v存储系统**
- 支持存储的value类型包括:String,list,set,zset,hash
- 这些数据类型都支持 push/pop、add/remove 及取交集并集和差集及更丰富的操作，而且这些操作都是原子性的
- 在此基础上，Redis 支持各种不同方式的排序
- 为了保证效率，数据都是缓存在内存中
- 区别的是 Redis 会周期性的把更新的数据写入磁盘或者把修改操作写入追加的记录文件(RDB & AOF)
- 并且在此基础上实现了 master-slave(主从)同步。



2.2 Redis安装

注：这里只讲Linux的安装过程

去官网，点击Download下载最新版本的zip文件

Download it

Redis 6.2.1 is the latest stable version.
 ou Interested in release candidates or
 ures unstable versions? [Check the](#)
[downloads page.](#)

2.2.1 安装步骤

准备工作：以CentOS7为例

1、下载最新版本的gcc编译器

```
yum install centos-release-scl scl-utils-build
yum install -y ddevtoolset-8-toolchain
scl enable devtoolset-8-bash
```

2、测试gcc版本

```
gcc --version
```

```
[bigdata@hdp1 redis-6.0.8]$ sudo gcc --version
gcc (GCC) 8.3.1 20190311 (Red Hat 8.3.1-3)
Copyright (C) 2018 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

3、采用xftp工具将windows上下载的redis压缩包放到自己在linux这边的目录

4、进入linux找到刚刚放redis压缩包的目录，打开终端

5、解压命令: `tar -zxvf redis-6.2.1.tar.gz`

6、解压完进入目录 `cd 解压目录`

7、在该目录执行 `make` (编译好)

8、如果在第一步时候没有准备C语言编译缓解, make会报错提示: -
Jemalloc/jemalloc.h:没有那个文件

解决方法: `make distclean`

8.1、在刚才解压目录下再次执行 `make` 命令

9、正式安装: `make install` 默认安装目录: `/usr/local/bin`

```
[root@localhost bin]# cd /usr/local/bin
[root@localhost bin]# ll
total 18916
-rw-r--r--. 1 root root      110 Mar 20 14:21 appendonly.aof
-rw-r--r--. 1 root root     118 Mar 20 15:24 dump.rdb
-rwxr-xr-x. 1 root root 4830528 Mar 18 11:24 redis-benchmark
lrwxrwxrwx. 1 root root      12 Mar 18 11:24 redis-check-aof -> redis-server
lrwxrwxrwx. 1 root root      12 Mar 18 11:24 redis-check-rdb -> redis-server
-rwxr-xr-x. 1 root root 5004856 Mar 18 11:24 redis-cli
lrwxrwxrwx. 1 root root      12 Mar 18 11:24 redis-sentinel -> redis-server
-rwxr-xr-x. 1 root root 9521760 Mar 18 11:24 redis-server
```

redis-benchmark: 性能测试工具, 可以在自己本子运行, 看看自己本子性能如何

redis-check-aof: 修复有问题的 AOF 文件

redis-check-dump: 修复有问题的 dump.rdb 文件

redis-sentinel: Redis 集群使用

redis-server: Redis 服务器启动命令

redis-cli: 客户端, 操作入口

2.3 启动Redis

2.3.1 前台启动 (不推荐)

前台启动, 命令行窗口不能关闭, 一旦关闭, redis服务停止 (ctrl + c 强制结束)

```
[root@zy bin]# redis-server
12471:C 16 Dec 12:33:08.178 # Warning: no config file specified, using the default config. In order
o specify a config file use redis-server /path/to/redis.conf
12471:M 16 Dec 12:33:08.178 * Increased maximum number of open files to 10032 (it was originally set
to 1024).

Redis 3.2.5 (00000000/0) 64 bit
Running in standalone mode
Port: 6379 默认端口号
PID: 12471

http://redis.io
```

2.3.2 后台启动（推荐）

一般建议将刚刚我们解压目录中的redis.conf配置文件 复制到另一个位置上

```
cp /myredis/redis-6.2.1/redis.conf /etc/redis.conf
```

修改配置文件redis.conf

```
@localhost bin]# vim /etc/redis.conf
```

后台启动设置 `daemonize no` 改成 `daemonize yes`

```
# By default Redis does not run as a daemon.
# Note that Redis will write a pid file.
# When Redis is supervised by upstart or
daemonize yes
```

后台启动redis服务

```
redis-server /etc/redis.conf
```

查看关于redis的后台进程

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

```
root      25351      1  0 14:24 ?        00:00:10 redis-server *:6379
root      26597  24915  0 16:02 pts/1    00:00:00 grep --color=auto redis
```

这里我的redis的地址是*:表示任意一台IP地址都可以访问，这需要在配置文件中配置

你们一开始查看的时候都是本机地址，即 `redis-server 127.0.0.1:6379`

采用客户端访问redis（必须先启动redis-server）

```
redis-cli
```

```
[root@localhost bin]# redis-cli  
127.0.0.1:6379> █
```

测试是否成功

在redis客户端输入 `ping`,如果出现 **PONG** , 恭喜你正式启动了Redis

Redis关闭

Linux命令行关闭 : `redis-cli shutdown`

redis客户端关闭 : `shutdown`

2.4 Redis 必知点

- Redis 默认端口号为 `6379`
- Redis 配置文件名称为: `redis.conf`
- Redis 默认16个数据库, 初始默认库位0号库
 - 切换库的指令为 `select x` `x: 0 ~ 15`
 - 统一密码管理, 所有库同样密码
 - 查看当前库key的数量 : `dbsize`
 - 清空当前库: `flushdb`
 - 通杀全部库(别用!): `flushall`
- Redis 采用 **单线程 + 多路I/O复用技术**
 - Redis的单线程指的是执行命令时的单线程, 体现在当客户端执行命令时, 一条命令从客户端到服务端不会立即被执行, 而是进入一个等待队列中, 每次只有一条指令被选中执行, 按顺序执行
 - 多路I/O复用技术用来解决I/O问题, 多路复用是指使用一个线程来检查多个文件描述符 (Socket) 的就绪状态, 比如调用 `select` 和 `poll` 函数, 传入多个文件描述符, 如果有一个文件描述符就绪, 则返回, 否则 阻塞直到超时。得到就绪状态后进行真正的操作可以在同一个线程里执行, 也可以启动线程执行 (比如使用线程池)

三、常用五大数据类型

- String(字符串)
- List(列表)
- Set(集合)
- Hash(哈希)
- Zset(有序集合)

3.1 键(Key)

`keys *` 查看当前库所有 key (匹配: `keys *1`)

`exists key` 判断某个 key 是否存在

`type key` 查看你的 key 是什么类型

`del key` 删除指定的 key 数据

`unlink key` 根据 value 选择非阻塞删除 仅将 keys 从 keyspace 元数据中删除，真正的删除会在后续异步操作。

`expire key seconds` 10 秒钟：为给定的 key 设置过期时间

`t1 key` 查看还有多少秒过期，-1 表示永不过期，-2 表示已过期

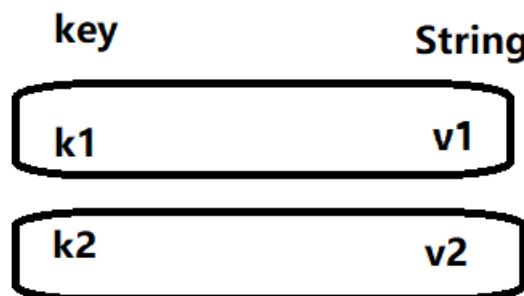
`select` 命令切换数据库

`dbsize` 查看当前数据库的 key 的数量

`flushdb` 清空当前库

`flushall` 通杀全部库

3.2 String字符串



String 是 Redis 最基本的类型 "一个 key 对应一个 value" String 类型是二进制安全的。意味着 Redis 的 string 可以包含任何数据。比如 jpg 图片 或者序列化的对象。

- 一个 Redis 中字符串 value 最多可以是 512M
- 键可以用a:b 方式增加键的提示信息

3.2.1 常用命令

- **set** 添加键值对

```
127.0.0.1:6379> set name wcd
OK
```

* `nx`: 当数据库key不存在时，可以将k-v添加到数据库

```
127.0.0.1:6379> setnx age 18
(integer) 1    //库中不存在，可以添加，返回1
127.0.0.1:6379> setnx name hhh
(integer) 0    //库中已存在，不可以添加，返回0
```

* `xx`(set): 当数据库中 key 存在时，可以将 key-value 添加数据库，与 nx 参数互斥

* `ex`: key 的超时秒数 setex

```
127.0.0.1:6379> setex day 20 day1
OK
127.0.0.1:6379> keys *
1) "day"
2) "age"
3) "name"
4) "k3"
5) "k1"
6) "k2"
.....20s后
127.0.0.1:6379> keys *
1) "age"
2) "name"
3) "k3"
4) "k1"
5) "k2"
```

* `px`: key 的超时毫秒数，与 ex 互斥

- **get** 查询给定key对应的value

```
127.0.0.1:6379> get name
"wcd"
```

- **append** 将给定的值追加到原值的末尾

```
127.0.0.1:6379> append name 666
(integer) 6 //返回追加后的字符串长度
127.0.0.1:6379> get name
"wdcd666"
```

- **strlen** 获得值的长度
- **incr** 将key中存储的数字增1，只能对数字值操作，如果为空，新增值为1
- **decr** 将key中存储的数字减1，只能对数字值操作，如果为空，新增值为-1
- **incrby/decrby** 将key中存储的数字值进行增减，自定义步长

```
127.0.0.1:6379> get age
"18"
127.0.0.1:6379> incrby age 20 //decrby同理
(integer) 38 //返回成功后的数字值
127.0.0.1:6379> get age
"38"
```

原子性

incr, decr, incrby, decrby具备原子性的操作叫做原子操作。所谓原子操作是指不会被线程调度机制打断的操作

这种操作一旦开始，就一直运行到结束，中间不会有任何 context switch（切换到另一个线程）。

(1) 在单线程中，能够在单条指令中完成的操作都可以认为是“原子操作”，因为中断只能发生于指令之间。

(2) 在多线程中，不能被其它进程（线程）打断的操作就叫原子操作。Redis 单命令的原子性主要得益于 Redis 的单线程。

- **mset** ... 同时设置一个或多个k-v对
- **mget** 同时获取一个或多个value
- **msetnx** ... 同时设置一个或多个k-v对，当且仅当所有给定的key都不存在，如果有任何一个存在，那么该指令操作失败(原子性：要么都成功，有一个失败就失败)
- **getrange** 获得值的范围，类似java中的subString

```
127.0.0.1:6379> mset k1 v1 k2 v2 k3 v3
OK
127.0.0.1:6379> mget k1 k2 k3
1) "v1"
2) "v2"
3) "v3"

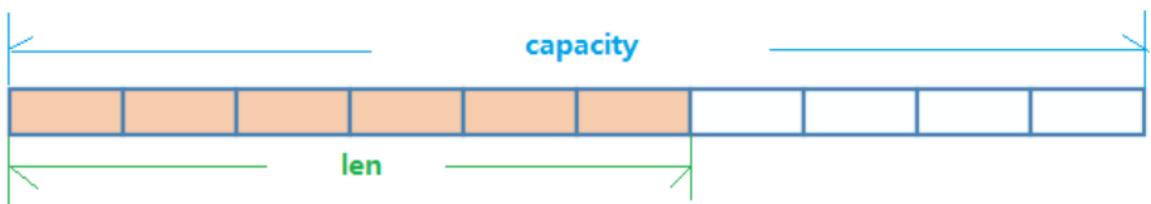
127.0.0.1:6379> getrange name 0 2
"wdcd" // "wdcd666" 从0开始到2截取字符串
```

- **setrange** 用覆盖所存储的字符串值，从开始(索引从0开始)
- **getset** 以旧换新，设置了新值的同时获得旧值

```
127.0.0.1:6379> getset name zpp
"wdcd666" //设置新值 "zpp" ,返回旧值 "wdcd666"
127.0.0.1:6379> get name
"zpp"
```

3.2.2 数据结构

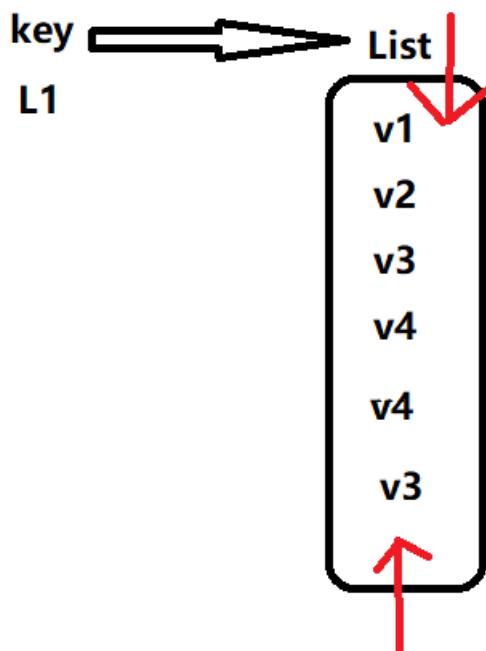
String 的数据结构为 简单动态字符串(Simple Dynamic String,缩写 SDS)。是可以修改的字符串，内部结构实现上类似于 Java 的 ArrayList，采用预分配冗余空间的方式来减少内存的频繁分配



如上图所示，内部为当前字符串实际分配的空间 capacity 一般要高于实际字符串长度 len。当字符串长度小于 1M 时，扩容都是加倍现有的空间，如果超过 1M，扩容时一次只会多扩 1M 的空间。需要注意的是字符串最大长度为 512M。

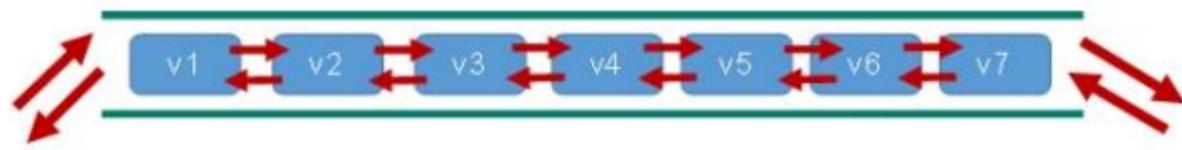
3.3 List列表

记住List：“**单键多值**”



Redis列表是简单的字符串列表，按照插入顺序排序。你可以添加一个元素到列表的头部(Left)或者尾部(Right)

底层是一个双向链表，对双端的操作性都很高，如果通过索引下标的操作方式，中间的节点性能会较差。



3.3.1 常用命令

List的命令基本都是L(List)开头

- **lpush/rpush** <value...> 从左边/右边插入一个或多个值。

```
127.0.0.1:6379> lpush number 1 2 3
(integer) 3
127.0.0.1:6379> rpush number 4 5 6
(integer) 6
127.0.0.1:6379> lrange number 0 -1
1) "3"
2) "2"
3) "1"
4) "4"
5) "5"
6) "6"
```

lpush(123) 3 2 1 | | | | --> 3 | 2 | 1

3 2 1 <---rpush(456) 3 2 1 4 5 6

lrange key 0 -1 :从左边下标为0的位置到最右边第一个下标-1 之间的所有value

- **lpop/rpop [count]** 从左边/右边吐出count个值。"值在键在，值光键亡"

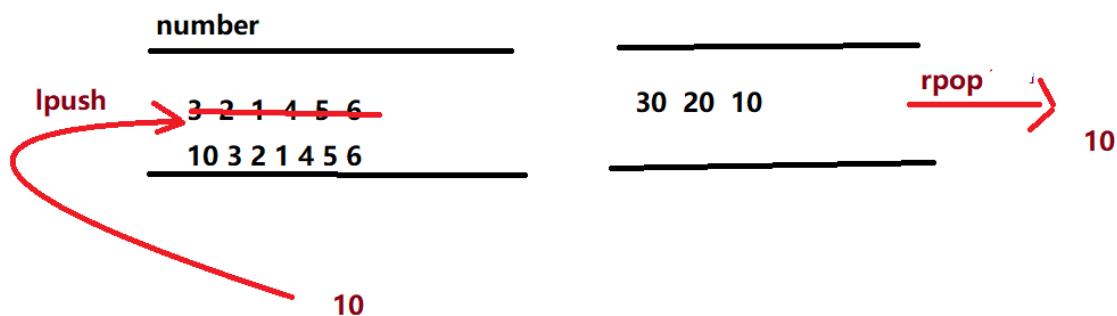
```
lpop number 1
1) "3"
```

- **rpoplpush** 从列表右边吐出一个值，插到列表左边。

```

127.0.0.1:6379> lpush size 10 20 30
(integer) 3
127.0.0.1:6379> rpoplpush size number
"10"
127.0.0.1:6379> lrange number 0 -1
1) "10"
2) "3"
3) "2"
4) "1"
5) "4"
6) "5"
7) "6"

```



- **lrange** 按照索引下标获得元素(从左到右) lrange mylist 0 -1 0 左边第一个， -1 右边第一个， (0-1 表示获取所有)
- **lindex** 按照索引下标获得元素(从左到右) 下标从0开始
- **llen** 获得列表长度
- **linsert** before/after 在的前面/后面插入

```

127.0.0.1:6379> lrange size 0 -1
1) "30"
2) "20"
3) "10"
127.0.0.1:6379> linsert size before 30 40
(integer) 4
127.0.0.1:6379> linsert size before 40 50
(integer) 5
127.0.0.1:6379> linsert size after 10 5
(integer) 6
127.0.0.1:6379> lrange size 0 -1
1) "50"
2) "40"
3) "30"
4) "20"
5) "10"
6) "5"

```

- **lrem** 从左边删除 n 个 value(从左到右)

```
127.0.0.1:6379> lrem size 2 10 //从左到右删除2个value为10的, 如果没有就不
删除, 不足2个也删除, 作用就是从左到右批量删除一样的值
(integer) 1
127.0.0.1:6379> lrange size 0 -1
1) "50"
2) "40"
3) "30"
4) "20"
5) "5"
```

- **lset** 将列表 key 下标为 index 的值替换成 value

```
127.0.0.1:6379> lset size 0 100 //将下标0的值替换成100
OK
127.0.0.1:6379> lrange size 0 -1
1) "100"
2) "40"
3) "30"
4) "20"
5) "5"
```

3.3.2 数据结构

List的数据结构为快速链表**QuickList**

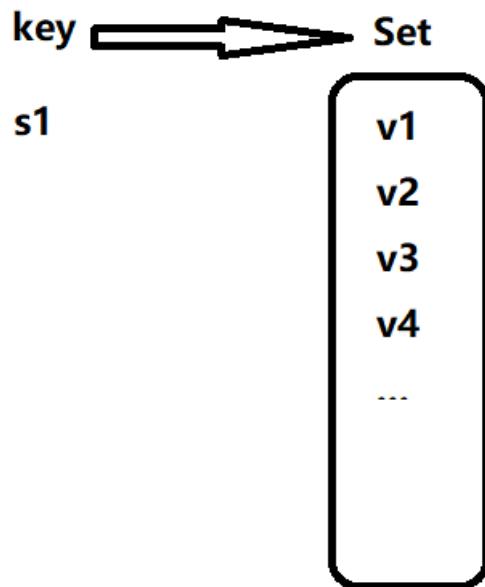
首先在列表元素较少的情况下会使用一块连续的内存存储，这个结构是 **ziplist**，也即是 压缩列表它将所有的元素紧挨着一起存储，分配的是一块连续的内存。当数据量比较多的时候才会改成 quicklist。

因为普通的链表需要的附加指针空间太大，会比较浪费空间。比如这个列表里存的只是 int 类型的数据，结构上还需要两个额外的指针 prev 和 next。



Redis 将链表和 ziplist 结合起来组成了 quicklist。也就是将多个 ziplist 使用双向指针串起来使用。这样既满足了快速的插入删除性能，又不会出现太大的空间冗余。

3.3 Set集合



“可以看作是有去重功能的List”

Redis set 对外提供的功能与 list 类似是一个列表的功能，特殊之处在于 **set 是可以自动排重**(类似于 MySql 的去重distinct)的，当你需要存储一个列表数据，又不希望出现重复数据时，set 是一个很好的选择。并且 set 提供了判断某个成员是否在一个 set 集合内的接口，这个也是 list 所不能提供的

Redis 的 Set 是 **String 类型的无序集合** (Zset与之相反)。它底层其实是一个 value 为 null 的 hash 表，所以添加，删除，查找的复杂度都是 O(1)。

一个算法，随着数据的增加，执行时间的长短，如果是 O(1)，数据增加，查找数据的时间不变

3.3.1 常用命令

Set的命令都以"s"开头

- **sadd** 将一个或多个 member 元素加入到集合 key 中，已经存在的 member 元素将被忽略

```
127.0.0.1:6379> sadd s1 v1 v2 v3 v1 v4 //排重，v1重复，只添加1个
(integer) 4
127.0.0.1:6379> smembers s1
1) "v1"
2) "v4"
3) "v3"
4) "v2"
127.0.0.1:6379> sismember s1 v1
(integer) 1
127.0.0.1:6379> scard s1
(integer) 4
127.0.0.1:6379> srem s1 v1 v2
```

```
(integer) 2
127.0.0.1:6379> smembers s1
1) "v4"
2) "v3"
127.0.0.1:6379> spop s1 1
1) "v3"
```

- **smembers** 取出该集合的所有值。
- **sismember** 判断集合是否为含有该值，有 1，没有 0
- **scard** 返回该集合的元素个数。
- **srem** 删除集合中的某元素。
- **spop** 随机从该集合中吐出n个值。
- **srandmember** 随机从该集合中取出 n 个值。不会从集合中删除。

```
127.0.0.1:6379> smembers s1
1) "v1"
2) "v2"
3) "v5"
4) "v3"
5) "v4"
127.0.0.1:6379> srandmember s1 2
1) "v1"
2) "v5"
127.0.0.1:6379> smove s1 s2 v1
(integer) 1
127.0.0.1:6379> smove s1 s2 v2
(integer) 1
127.0.0.1:6379> smembers s2
1) "v1"
2) "v2"
```

- **smove** 把集合A中的一个值从移动到集合B
- **sinter** 返回两个集合的交集元素。
- **sunion** 返回两个集合的并集元素。
- **sdiff** 返回两个集合的差集元素(key1 中的，不包含 key2 中的)

3.3.2 数据结构

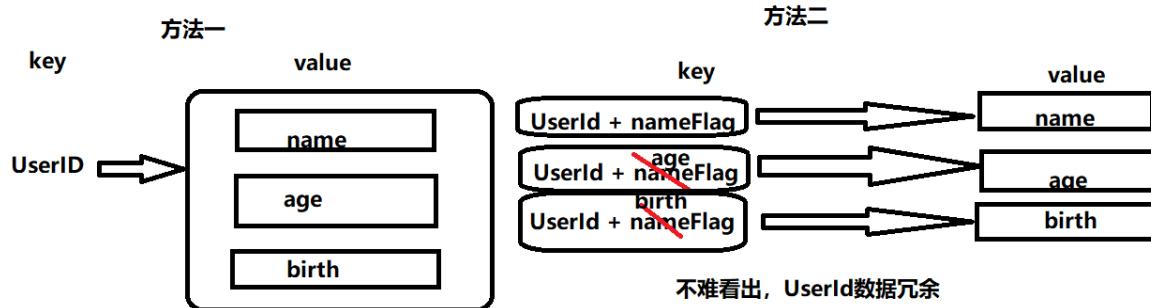
Set 数据结构是 **dict 字典**，字典是用**哈希表**实现的。Java 中 HashSet 的内部实现使用的是 HashMap，只不过所有的 value 都指向同一个对象。Redis 的 set 结构也是一样，它的内部也使用 hash 结构，所有的 value 都指向同一个内部值。

3.4 Hash哈希

Redis hash 是一个键值对集合

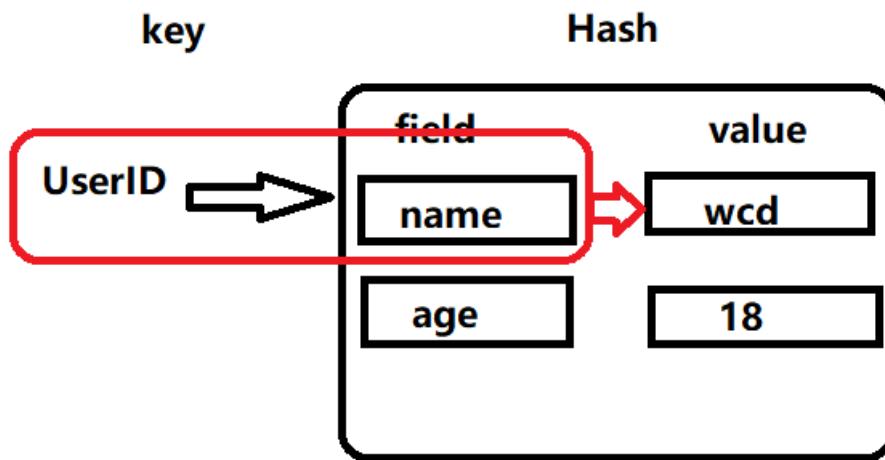
Redis hash 是一个 **String 类型的 field 和 value 的映射表**, hash 特别适合用于存储对象。类似 Java 里面的 Map<String, Object>

假设要存储一个用户, 用户 ID 为查找的 key, 存储的 value 用户对象包含姓名, 年龄, 生日等信息, 如果用普通的 key/value 结构来存储, 主要有两种方式



每次修改用户的某个属性, 需要取出每个属性的值, 开销较大

而Hash哈希就是上面两种方法的折中, 通过key(UserID) + field(属性标签)就可以操作对应的属性数据, 即不需要重读存储数据, 也不会带来序列化和并发修改控制的问题



key -> Hash(field -> value)

3.4.1 常用命令

和上面一样, Hash的命令都是"h"开头

- **hset** ... 给集合中的键赋值
- **hget** 从集合取出 value
- **hmset** .. 批量设置 hash 的值
- **hmget**... 查看hash中field对应的值
- **hexists** 查看哈希表 key 中, 给定域 field 是否存在。返回1表示存在
- **hkeys** 列出该 hash 集合的所有 field
- **hvals** 列出该 hash 集合的所有 value
- **hincrby** 为哈希表 key 中的域 field 的值加上增量 1 -1

- **hsetnx** 将哈希表 key 中的域 field 的值设置为 value , 当且仅当域 field 不存在

```

127.0.0.1:6379> hset user:1001 name wcd age 18 sex male
(integer) 3
127.0.0.1:6379> hget user:1001 name
"wcd"
127.0.0.1:6379> hget user:1001 age
"18"
127.0.0.1:6379> hget user:1001 sex
"male"
127.0.0.1:6379> hmset user:1002 name zpp age 18 sex female
OK
127.0.0.1:6379> hmget user:1002 name age sex
1) "zpp"
2) "18"
3) "female"
127.0.0.1:6379> hkeys user:1001
1) "name"
2) "age"
3) "sex"
127.0.0.1:6379> hvals user:1001
1) "wcd"
2) "18"
3) "male"
127.0.0.1:6379> hincrby user:1001 age 2
(integer) 20
127.0.0.1:6379> hincrby user:1001 age -2
(integer) 18
127.0.0.1:6379> hsetnx user:1001 name wll //name存在, 无法重新赋值, 返回
0
(integer) 0
127.0.0.1:6379> hsetnx user:1001 inters basketball
(integer) 1

```

3.4.2 数据结构

Hash 类型对应的数据结构是两种：**ziplist (压缩列表)**，**hashtable (哈希表)**。当 field-value 长度较短且个数较少时，使用 ziplist，否则使用 hashtable。

3.5 Zset有序集合

Redis 有序集合 zset 与普通集合 set 非常相似，是一个**没有重复元素的字符串集合**。

不同之处是有序集合的每个成员都关联了一个评分(score),这个评分(score)被用 来按照从最低分到最高分的方式排序集合中的成员。集合的成员是唯一的，但是评分可以是重复的。

因为元素是有序的, 所以可以很快的根据评分(score) 或者次序(position) 来获取一个范围的元素。访问有序集合的中间元素也是非常快的,因此你能够使用有序集合作为一个没有重复成员的智能列表。

3.5.1 常用命令

- **zadd** ... 将一个或多个 member 元素及其 score 值加入到有序集 key 当中。
- **zrange [WITHSCORES]** 返回有序集 key 中, 下标在之间的元素 带 WITHSCORES, 可以让分数一起和值返回到结果集。 与其相反的是zrevrange (reverse反转)
- **zrangebyscore [withscores] [limit offset count]** 返回有序集 key 中, 所有 score 值介于 min 和 max 之间(包括等于 min 或 max)的成员。 有序集成员按 score 值递增(从小到大)次序排列。
- **zrevrangebyscore [withscores] [limit offset count]** 同上, 改为从大到小排列。
- **zincrby** 为元素的 score 加上增量
- **zrem** 删除该集合下, 指定值的元素
- **zcount** 统计该集合, 分数区间内的元素个数
- **zrank** 返回该值在集合中的排名, 从 0 开始。

```
127.0.0.1:6379> zadd scores 100 math 90 english 95 chinese
(integer) 3
127.0.0.1:6379> zrange scores 0 -1 //从小到大
1) "english"
2) "chinese"
3) "math"
127.0.0.1:6379> zrevrange scores 0 -1 //从大到小
1) "math"
2) "chinese"
3) "english"
127.0.0.1:6379> zrange scores 0 -1 withscores //带分数
1) "english"
2) "90"
3) "chinese"
4) "95"
5) "math"
6) "100"
127.0.0.1:6379> zrevrange scores 0 -1 withscores
1) "math"
2) "100"
3) "chinese"
4) "95"
5) "english"
```

```

6) "90"
127.0.0.1:6379> zrangebyscore scores 95 100 withscores //根据分数由小到大排序, 从95-100
1) "chinese"
2) "95"
3) "math"
4) "100"
127.0.0.1:6379> zrevrangebyscore scores 100 95 withscores //根据分数由大到小排序, 从100-95
1) "math"
2) "100"
3) "chinese"
4) "95"
127.0.0.1:6379> zincrby scores 2 english
"92"

127.0.0.1:6379> zrem scores english
(integer) 1
127.0.0.1:6379> zrange scores 0 -1
1) "chinese"
2) "math"

```

- 案例：如何利用 zset 实现一个文章访问量的排行榜？

采用zrevrange 根据分数排序

```

127.0.0.1:6379> zadd booktop java 100 c 99 python 90 javascript 95
(error) ERR value is not a valid float
127.0.0.1:6379> zadd booktop 1000 java 500 python 600 javascript
(integer) 3
127.0.0.1:6379> zrevrange booktop 0 -1 withscores
1) "java"
2) "1000"
3) "javascript"
4) "600"
5) "python"
6) "500"

```

3.5.2 数据结构

SortedSet(zset)是 Redis 提供的一个非常特别的数据结构，**一方面它等价于 Java 的数据结构 Map**，可以给每一个元素 value 赋予一个权重 score，另一方面它**又类似于 TreeSet**，内部的元素会按照权重 score 进行排序，可以得到每个元素的名次，还可以通过 score 的范围来获取元素的列表。

zset 底层使用了两个数据结构

(1) **hash**，hash 的作用就是关联元素 value 和权重 score，保障元素 value 的唯一性，可以通过元素 value 找到相应的 score 值。

(2) 跳跃表，跳跃表的目的在于给元素 value 排序，根据 score 的范围获取元素列表。

3.5.2.1 跳跃表

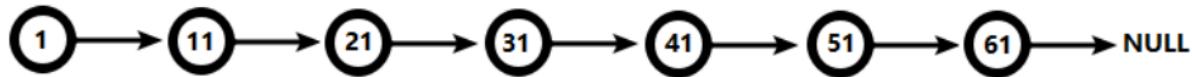
跳跃表又称跳表

有序集合在生活中比较常见，例如根据成绩对学生排名，根据得分对玩家排名等。对于有序集合的底层实现，可以用数组、平衡树、链表等。数组对于元素的插入、删除比较繁琐；平衡树或红黑树虽然效率高但结构复杂；链表查询遍历所有元素时效率低。Redis 采用的是跳跃表。跳跃表效率堪比红黑树，实现远比红黑树简单。

例子：

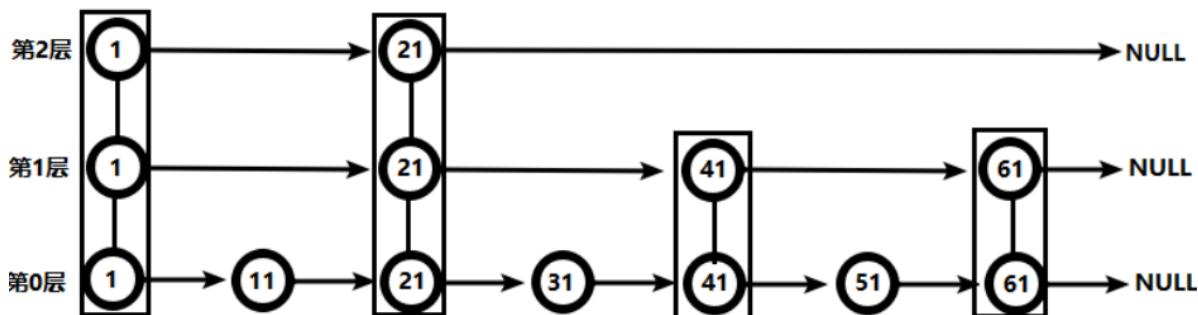
对比有序链表和跳跃表，从链表中查询出 51

1) 有序链表



要查找值为 51 的元素，需要从第一个元素开始依次查找、比较才能找到。共需要 6 次比较。

2) 跳表



从第 2 层开始，1 节点比 51 节点小，向后比较。

21 节点比 51 节点小，继续向后比较，后面就是 NULL 了，所以从 21 节点向下到第 1 层

在第 1 层，41 节点比 51 节点小，继续向后，61 节点比 51 节点大，所以从 41 向下

在第 0 层，51 节点为要查找的节点，节点被找到，共查找 4 次。

从此可以看出跳跃表比有序链表效率要高

四、Redis 配置文件

安装完的redis目录中的conf文件我复制到了 /etc/redis.conf

```
[root@localhost bin]# cd /home/lebrwcd/opt/redis-6.2.6/
[root@localhost redis-6.2.6]# ll
total 248
-rw-rw-r--. 1 root root 33624 Oct  4 18:59 00-RELEASENOTES
-rw-rw-r--. 1 root root     51 Oct  4 18:59 BUGS
-rw-rw-r--. 1 root root  5026 Oct  4 18:59 CONDUCT
-rw-rw-r--. 1 root root  3384 Oct  4 18:59 CONTRIBUTING
-rw-rw-r--. 1 root root  1487 Oct  4 18:59 COPYING
drwxrwxr-x. 7 root root  213 Mar 18 11:23 deps
-rw-r--r--. 1 root root    92 Mar 19 15:01 dump.rdb
-rw-rw-r--. 1 root root    11 Oct  4 18:59 INSTALL
-rw-rw-r--. 1 root root   151 Oct  4 18:59 Makefile
-rw-rw-r--. 1 root root  6888 Oct  4 18:59 MANIFESTO
-rw-rw-r--. 1 root root 21567 Oct  4 18:59 README.md
-rw-rw-r--. 1 root root 93725 Mar 19 15:48 redis.conf
-rwxrwxr-x. 1 root root   275 Oct  4 18:59 runtest
-rwxrwxr-x. 1 root root   279 Oct  4 18:59 runtest-cluster
-rwxrwxr-x. 1 root root  1079 Oct  4 18:59 runtest-moduleapi
-rwxrwxr-x. 1 root root   281 Oct  4 18:59 runtest-sentinel
-rw-rw-r--. 1 root root 13768 Oct  4 18:59 sentinel.conf
drwxrwxr-x. 3 root root 12288 Mar 18 11:24 src
drwxrwxr-x. 11 root root  182 Oct  4 18:59 tests
-rw-rw-r--. 1 root root  3055 Oct  4 18:59 TLS.md
drwxrwxr-x. 9 root root  4096 Oct  4 18:59 utils
```

```
[root@localhost /]# cd /etc
[root@localhost etc]# find redis.*  
redis.conf
[root@localhost etc]#
```

让我们从上到下看看配置文件

4.1 Units单位

```
# Redis configuration file example.
#
# Note that in order to read the configuration file, Redis must be
# started with the file path as first argument:
#
# ./redis-server /path/to/redis.conf
#
# Note on units: when memory size is needed, it is possible to specify
# it in the usual form of 1k 5GB 4M and so forth:
# Redis configuration file example.
#
# Note that in order to read the configuration file, Redis must be
# started with the file path as first argument:
#
# ./redis-server /path/to/redis.conf
#
# Note on units: when memory size is needed, it is possible to specify
# it in the usual form of 1k 5GB 4M and so forth:
#
# 1k => 1000 bytes
# 1kb => 1024 bytes
# 1m => 1000000 bytes
# 1mb => 1024*1024 bytes
# 1g => 1000000000 bytes
# 1gb => 1024*1024*1024 bytes
#
# units are case insensitive so 1GB 1Gb 1gB are all the same.
#####
##### INCLUDES #####
```

- 配置了大小单位，只支持bytes，不支持bit
- 大小写不敏感 (units are case insensitive so 1GB 1Gb 1gB are all the same)

4.2 includes包含

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

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

类似于JSP中的include，多实例的情况可以把公用的配置文件提取出来，然后包含到其他配置文件

4.3 网络相关配置

```
#####
##### NETWORK #####
####

# By default, if no "bind" configuration directive is specified, Redis listens
# for connections from all available network interfaces on the host machine.
# It is possible to listen to just one or multiple selected interfaces using
# the "bind" configuration directive, followed by one or more IP addresses.
# Each address can be prefixed by "-", which means that redis will not fail to
# start if the address is not available. Being not available only refers to
# addresses that does not correspond to any network interface. Addresses that
# are already in use will always fail, and unsupported protocols will always BE
# silently skipped.
#
# Examples:
#
# bind 192.168.1.100 10.0.0.1      # listens on two specific IPv4 addresses
# bind 127.0.0.1 ::1                # listens on loopback IPv4 and IPv6
# bind * -::*                        # like the default, all available interfaces
#
# ~~~ WARNING ~~~ If the computer running Redis is directly exposed to the
# internet, binding to all the interfaces is dangerous and will expose the
# instance to everybody on the internet. So by default we uncomment the
# following bind directive, that will force Redis to listen only on the
# IPv4 and IPv6 (if available) loopback interface addresses (this means Redis
# will only be able to accept client connections from the same host that it is
# running on).
#
# IF YOU ARE SURE YOU WANT YOUR INSTANCE TO LISTEN TO ALL THE INTERFACES
```

4.3.1 bind

```

# will only be able to accept client connections from the same host that it is
# running on).
#
# IF YOU ARE SURE YOU WANT YOUR INSTANCE TO LISTEN TO ALL THE INTERFACES
# JUST COMMENT OUT THE FOLLOWING LINE.
# ~~~~~
#bind 127.0.0.1 -::1 不修改的情况下是没注释的，也就是说默认只
# 接收本机的访问请求
# Protected mode is a layer of security protection, in order to avoid that
# Redis instances left open on the internet are accessed and exploited.
#
# When protected mode is on and if:
#
# 1) The server is not binding explicitly to a set of addresses using the
#    "bind" directive.
# 2) No password is configured.
#
# The server only accepts connections from clients connecting from the
# IPv4 and IPv6 loopback addresses 127.0.0.1 and ::1, and from Unix domain
# sockets.
#
# By default protected mode is enabled. You should disable it only if
# you are sure you want clients from other hosts to connect to Redis
# even if no authentication is configured, nor a specific set of interfaces
# are explicitly listed using the "bind" directive.
protected-mode no 保护策略

# Accept connections on the specified port, default is 6379 (IANA #815344).
# If port 0 is specified Redis will not listen on a TCP socket.

```

- 默认情况 bind=127.0.0.1 只能接受本机的访问请求
- 不写的情况下，无限制接受任何 ip 地址的访问
- 生产环境肯定要写你应用服务器的地址；服务器是需要远程访问的，所以需要将其注释掉 如果开启了 protected-mode，那么在没有设定 bind ip 且没有设密码的情况下，Redis 只允许接受本机的响应
- 保存配置，停止服务，重启启动查看进程，不再是本机访问了。
 - 如何停止服务？

- 客户端->`shutdown` / linux终端 -->`redis-cli shutdown` / 强制杀死进程 `kill -9 pid`
- 之后重启服务 `redis-server /etc/redis.conf`
- 查看进程 `ps -ef | grep redis`

```

dis]# ps -ef|grep redis
49      1  0 16:23 ?        00:00:00 redis-server *:6379
53  12006  0 16:23 pts/2    00:00:00 grep --color=auto redi

```

4.3.2 protected-mode

将本机访问保护模式设置 no,不然就算只改了bind Id，远程ip也无法访问，后续可能还要关闭防火墙

4.3.3 Port

端口号，默认 6379

```

# Accept connections on the specified port, default is 6379 (IANA #815344)
# If port 0 is specified Redis will not listen on a TCP socket.
port 6379

```

4.3.4 tcp-backlog

```
# TCP listen() backlog.
#
# In high requests-per-second environments you need a high backlog in order
# to avoid slow clients connection issues. Note that the Linux kernel
# will silently truncate it to the value of /proc/sys/net/core/somaxconn so
# make sure to raise both the value of somaxconn and tcp_max_syn_backlog
# in order to get the desired effect.
tcp-backlog 511
```

- 设置 tcp 的 backlog, backlog 其实是一个连接队列, backlog 队列总和=未完成三次握手队列 + 已经完成三次握手队列。
- 在高并发环境下你需要一个高 backlog 值来避免慢客户端连接问题。
- 注意 Linux 内核会将这个值减小到/proc/sys/net/core/somaxconn 的值 (128) , 所以需要确认增大/proc/sys/net/core/somaxconn 和/proc/sys/net/ipv4/tcp_max_syn_backlog (128) 两个值来达到想要的效果

4.3.5 timeout

- 一个空闲的客户端维持多少秒会关闭, 0 表示关闭该功能。即永不关闭。

```
# Close the connection after a client is idle for N seconds (0 to disable)
timeout 0
```

4.3.6 tcp-keepalive

- 对访问客户端的一种心跳检测, 每个 n 秒检测一次。单位为秒, 如果设置为 0, 则不会进行 Keepalive 检测, 建议设置成 60

```
# TCP keepalive.
#
# If non-zero, use S0_KEEPALIVE to send TCP ACKs to clients in absence
# of communication. This is useful for two reasons:
#
# 1) Detect dead peers.
# 2) Force network equipment in the middle to consider the connection to be
#    alive.
#
# On Linux, the specified value (in seconds) is the period used to send ACKs.
# Note that to close the connection the double of the time is needed.
# On other kernels the period depends on the kernel configuration.
#
# A reasonable value for this option is 300 seconds, which is the new
# Redis default starting with Redis 3.2.1.
tcp-keepalive 300
```

4.4 通用配置

```
#####
# GENERAL #####
####

# By default Redis does not run as a daemon. Use 'yes' if you need it.
#a Note that Redis will write a pid file in /var/run/redis.pid when daemonized.
# When Redis is supervised by upstart or systemd, this parameter has no impact.
daemonize yes

# If you run Redis from upstart or systemd, Redis can interact with your
# supervision tree. Options:
# supervised no      - no supervision interaction
# supervised upstart - signal upstart by putting Redis into SIGSTOP mode
#                      requires "expect stop" in your upstart job config
# supervised systemd - signal systemd by writing READY=1 to $NOTIFY_SOCKET
#                      on startup, and updating Redis status on a regular
#                      basis.
# supervised auto   - detect upstart or systemd method based on
#                      UPSTART_JOB or NOTIFY_SOCKET environment variables
# Note: these supervision methods only signal "process is ready."
#       They do not enable continuous pings back to your supervisor.
#
# The default is "no". To run under upstart/systemd, you can simply uncomment
# the line below:
#
# supervised auto

# If a pid file is specified, Redis writes it where specified at startup
# and removes it at exit.
#
# When the server runs non daemonized, no pid file is created if none is
```

4.4.1 daemonize

- 是否为后台进程，设置为 yes 守护进程，后台启动

```
# By default Redis does not run as a daemon. Use 'yes' if you need it.
#a Note that Redis will write a pid file in /var/run/redis.pid when daemonized.
# When Redis is supervised by upstart or systemd, this parameter has no impact.
daemonize yes
```

4.4.2 pidfile

- 存放 pid 文件的位置，每个实例会产生一个不同的 pid 文件

```
#
# Creating a pid file is best effort: if Redis is not able to create it
# nothing bad happens, the server will start and run normally.
#
# Note that on modern Linux systems "/run/redis.pid" is more conforming
# and should be used instead.
pidfile /var/run/redis_6379.pid
```

4.4.3 logfile

- 日志文件名称

```
# Specify the log file name. Also the empty string can be used to force
# Redis to log on the standard output. Note that if you use standard
# output for logging but daemonize, logs will be sent to /dev/null
logfile ""

# To enable logging to the custom logger, just set 'loglevel enabled' to yes
```

4.4.4 loglevel

- 指定日志记录级别，Redis总共支持四个级别：debug,verbose,notice,warning，
默认为Notice

四个级别根据使用阶段来选择，生产环境选择 notice 或者 warning

4.4.4 databases 16

- 设定库的数量 默认 16， 默认数据库为 0， 可以使用 SELECT 命令在连接上指定数据库

```
# Set the number of databases. The default database is DB 0, you can select
# a different one on a per-connection basis using SELECT <dbid> where
# dbid is a number between 0 and 'databases'-1
databases 16
```

4.5 安全配置

```
#####
# SECURITY #####
#####

# Warning: since Redis is pretty fast, an outside user can try up to
# 1 million passwords per second against a modern box. This means that you
# should use very strong passwords, otherwise they will be very easy to break.
# Note that because the password is really a shared secret between the client
# and the server, and should not be memorized by any human, the password
# can be easily a long string from /dev/urandom or whatever, so by using a
# long and unguessable password no brute force attack will be possible.

# Redis ACL users are defined in the following format:
#
#   user <username> ... acl rules ...
#
# For example:
#
#   user worker +@list +@connection ~jobs:* on >ffa9203c493aa99
#
# The special username "default" is used for new connections. If this user
# has the "nopass" rule, then new connections will be immediately authenticated
# as the "default" user without the need of any password provided via the
# AUTH command. Otherwise if the "default" user is not flagged with "nopass"
# the connections will start in not authenticated state, and will require
# AUTH (or the HELLO command AUTH option) in order to be authenticated and
# start to work.
#
# The ACL rules that describe what a user can do are the following:
```

4.5.1 设置密码

```
# IMPORTANT NOTE: starting with Redis 6 "requirepass" is just a compatibility
# layer on top of the new ACL system. The option effect will be just setting
# the password for the default user. Clients will still authenticate using
# AUTH <password> as usually, or more explicitly with AUTH default <password>
# if they follow the new protocol: both will work.
#
# The requirepass is not compatible with aclfile option and the ACL LOAD
# command, these will cause requirepass to be ignored.
#
# requirepass foobared
```

- 访问密码的查看、设置和取消
- 在命令中设置密码，只是临时的。重启 redis 服务器，密码就还原了。
- 永久设置，需要再配置文件中进行设置

```
127.0.0.1:6379> config get requirepass
1) "requirepass"
2) ""
127.0.0.1:6379> config set requirepass "123456"
OK
127.0.0.1:6379> config get requirepass
(error) NOAUTH Authentication required.
127.0.0.1:6379> auth 123456
...
```

4.6 限制配置

4.6.1 maxclients

```
# Set the max number of connected clients at the same time. By default
# this limit is set to 10000 clients, however if the Redis server is not
# able to configure the process file limit to allow for the specified limit
# the max number of allowed clients is set to the current file limit
# minus 32 (as Redis reserves a few file descriptors for internal uses).
#
# Once the limit is reached Redis will close all the new connections sending
# an error 'max number of clients reached'.
#
# IMPORTANT: When Redis Cluster is used, the max number of connections is also
# shared with the cluster bus: every node in the cluster will use two
# connections, one incoming and another outgoing. It is important to size the
# limit accordingly in case of very large clusters.
#
# maxclients 10000
```

- 设置 redis 同时可以与多少个客户端进行连接。
- 默认情况下为 10000 个客户端。
- 如果达到了此限制，redis 则会拒绝新的连接请求，并且向这些连接请求方发出“max number of clients reached”以作回应。

4.6.2 maxmemory

- 建议必须设置，否则，将内存占满，造成服务器宕机
- 设置 redis 可以使用的内存量。一旦到达内存使用上限，redis 将会试图移除内部数据，移除规则可以通过 maxmemory-policy 来指定。
- 如果 redis 无法根据移除规则来移除内存中的数据，或者设置了“不允许移除”，那么 redis 则会针对那些需要申请内存的指令返回错误信息，比如 SET、LPUSH 等。
- 但是对于无内存申请的指令，仍然会正常响应，比如 GET 等。如果你的 redis 是主 redis (说明你的 redis 有从 redis)，那么在设置内存使用上限时，需要在系统中留出一些内存空间给同步队列缓存，只有在你设置的是“不移除”的情况下，才不用考虑这个因素

```
#####
# MEMORY MANAGEMENT #####
#
# Set a memory usage limit to the specified amount of bytes.
# When the memory limit is reached Redis will try to remove keys
# according to the eviction policy selected (see maxmemory-policy).
#
# If Redis can't remove keys according to the policy, or if the policy is
# set to 'noeviction', Redis will start to reply with errors to commands
# that would use more memory, like SET, LPUSH, and so on, and will continue
# to reply to read-only commands like GET.
#
# This option is usually useful when using Redis as an LRU or LFU cache, or to
# set a hard memory limit for an instance (using the 'noeviction' policy).
#
# WARNING: If you have replicas attached to an instance with maxmemory on,
# the size of the output buffers needed to feed the replicas are subtracted
# from the used memory count, so that network problems / resyncs will
# not trigger a loop where keys are evicted, and in turn the output
# buffer of replicas is full with DEls of keys evicted triggering the deletion
# of more keys, and so forth until the database is completely emptied.
#
# In short... if you have replicas attached it is suggested that you set a lower
# limit for maxmemory so that there is some free RAM on the system for replica
# output buffers (but this is not needed if the policy is 'noeviction').
#
# maxmemory <bytes>
#
# MAXMEMORY POLICY: how Redis will select what to remove when maxmemory
# is reached. You can select one from the following behaviors:
```

4.6.3 maxmemory-policy

- volatile-lru：使用 LRU 算法移除 key，只对设置了过期时间的键；（最近最少使用）
- allkeys-lru：在所有集合 key 中，使用 LRU 算法移除 key
- volatile-random：在过期集合中移除随机的 key，只对设置了过期时间的键
- allkeys-random：在所有集合 key 中，移除随机的 key
- volatile-ttl：移除那些 TTL 值最小的 key，即那些最近要过期的 key
- noeviction：不进行移除。针对写操作，只是返回错误信息

```
# MAXMEMORY POLICY: how Redis will select what to remove when maxmemory  
# is reached. You can select one from the following behaviors:  
#  
# volatile-lru -> Evict using approximated LRU, only keys with an expire set.  
# allkeys-lru -> Evict any key using approximated LRU.  
# volatile-lfu -> Evict using approximated LFU, only keys with an expire set.  
# allkeys-lfu -> Evict any key using approximated LFU.  
# volatile-random -> Remove a random key having an expire set.  
# allkeys-random -> Remove a random key, any key.  
# volatile-ttl -> Remove the key with the nearest expire time (minor TTL)  
# noeviction -> Don't evict anything, just return an error on write operations.  
  
#  
# LRU means Least Recently Used  
# LFU means Least Frequently Used  
#  
# Both LRU, LFU and volatile-ttl are implemented using approximated  
# randomized algorithms.  
  
# Note: with any of the above policies, when there are no suitable keys for  
# eviction, Redis will return an error on write operations that require  
# more memory. These are usually commands that create new keys, add data or  
# modify existing keys. A few examples are: SET, INCR, HSET, LPUSH, SUNIONSTORE,  
# SORT (due to the STORE argument), and EXEC (if the transaction includes any  
# command that requires memory).
```

####

4.6.4 maxmemory-samples

- 设置样本数量，LRU 算法和最小 TTL 算法都并非是精确的算法，而是估算值，所以你可以设置样本的大小，redis 默认会检查这么多个 key 并选择其中 LRU 的那个。
- 一般设置 3 到 7 的数字，数值越小样本越不准确，但性能消耗越小。

```
#  
# The default of 5 produces good enough results. 10 Approximates very closely  
# true LRU but costs more CPU. 3 is faster but not very accurate.  
#  
# maxmemory-samples 5  
  
# Eviction processing is designed to function well with the default setting.  
# If there is an unusually large amount of write traffic, this value may need to  
# be increased. Decreasing this value may reduce latency at the risk of  
# eviction processing effectiveness  
# 0 = minimum latency, 10 = default, 100 = process without regard to latency  
#
```

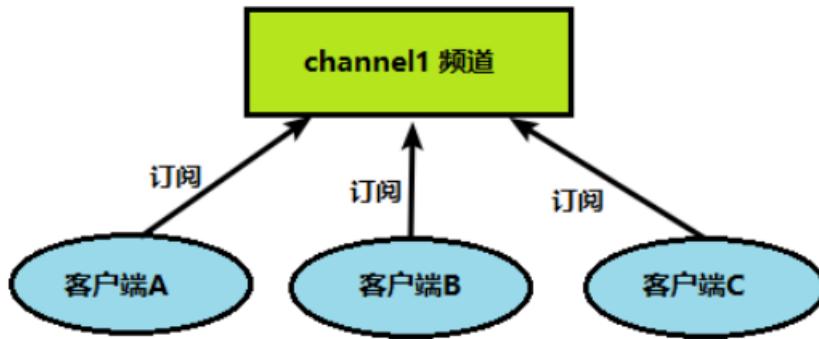
五、Redis的发布与订阅

5.1 什么是发布与订阅

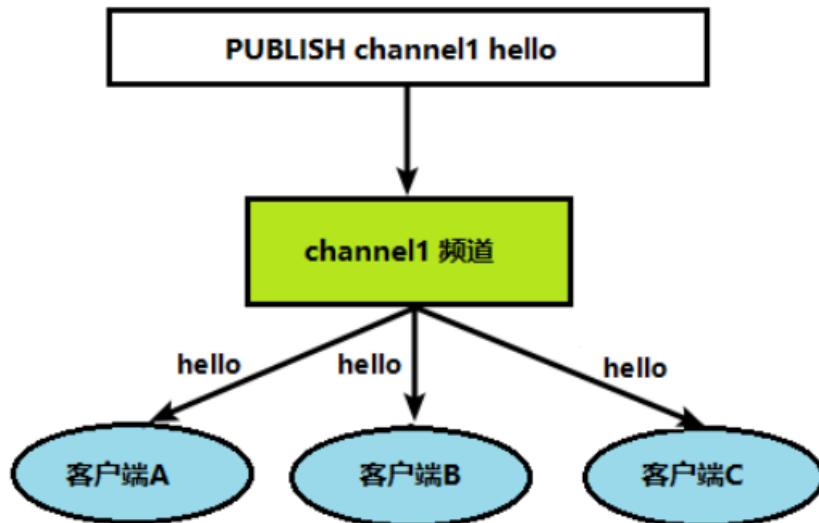
- Redis中的(pub/sub)发布与订阅是一种消息通信模式；发送者(publish)发送消息，订阅者(subscriber)接收消息
- Redis 客户端可以订阅任意数量的频道

5.2 图片演示

1. 客户端可以订阅频道如下图



2. 当给这个频道发布消息后，消息就会发送给订阅的客户端



5.3 发布订阅命令行实现

1. 打开一个redis客户端充当订阅者去订阅频道channel

```
subscriber channel1
```

```
127.0.0.1:6379> subscribe channel1
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "channel1"
3) (integer) 1
```

2. 打开另一个客户端(这里指的是多窗口打开，不是在一个终端中)，充当发布者



```
publish channel1 hello
```

```
127.0.0.1:6379> publish channel1 hello
(integer) 1
```

返回的1是订阅者的数量

3. 打开第一个客户端可以看到发布者发送的消息

```
127.0.0.1:6379> subscribe channel1
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "channel1"
3) (integer) 1
1) "message"
2) "channel1"
3) "hello"
```

注：发布的消息没有持久化

六、新数据类型

6.1 Bitmaps位图

6.2 HyperLoglog基数集

6.3 Geospatial地理空间

七、Jedis操作Redis

```
<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
    <version>3.2.0</version>
</dependency>
```

八、SpringBoot集成Redis

在pom中加入依赖性：

```

<!--
https://mvnrepository.com/artifact/org.springframework.boot/spring-
boot-starter-data-redis -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
    <version>2.6.2</version>
</dependency>

<!-- https://mvnrepository.com/artifact/redis.clients/jedis --
>
<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
    <version>2.9.0</version>
</dependency>

```

先测试一下能不能连通Jedis

1. redis.conf 中的 bind注释掉, protected-mode no
2. 关掉防火墙
3. 重启redis-server
4. ping

```

@Test
void contextLoads() {

    Jedis jedis = new Jedis("192.168.110.131", 6379);
    String ping = jedis.ping(); //出现pong表示连接成功

    System.out.println(ping);
    jedis.set("hello", "wcd");
}

```

配置文件:

`spring.redis`开头

```

spring:
  redis:
    jedis:
      pool:
        enabled: true
      port: 6379
      host: 192.168.110.131
      database: 0
      connect-timeout: 1800000
    lettuce:

```

```
pool:  
    max-active: 20  
    max-wait: -1  
    max-idle: 5  
    min-idle: 0
```

九、Redis事务、锁(秒杀案例)

9.x multi exec discard

9.x lua 脚步

十、Redis持久化

10.1 RDB

10.2 AOF

十一、主从复制

1、创建/myredis文件夹

2、复制redis.conf配置文件到文件夹中

3、配置一主两从，创建三个配置文件

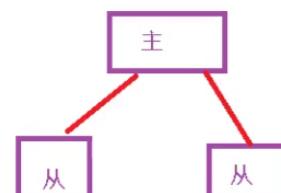
* redis6379.conf

* redis6380.conf

* redis6381.conf

4、在三个配置文件写入内容

```
include /myredis/redis.conf  
pidfile /var/run/redis_6379.pid  
port 6379  
dbfilename dump6379.rdb
```



5、启动三个redis服务

* 查看当前主机运行状况

* 在从机上执行 slaveof 主机ip 端口号

```
[lebrwcd@localhost ~]$ mkdir myredis      pwd之后完整路径:  
[lebrwcd@localhost ~]$ ll                  /home/lebrwcd/myredis  
total 0  
drwxr-xr-x. 2 lebrwcd lebrwcd 6 Feb 11 20:54 Desktop  
drwxr-xr-x. 2 lebrwcd lebrwcd 6 Feb 11 20:54 Documents  
drwxr-xr-x. 2 lebrwcd lebrwcd 6 Feb 11 20:54 Downloads  
drwxr-xr-x. 2 lebrwcd lebrwcd 6 Feb 11 20:54 Music  
drwxrwxr-x. 2 lebrwcd lebrwcd 6 Mar 21 16:06 myredis  
drwxr-xr-x. 4 lebrwcd lebrwcd 202 Mar 18 11:30 opt  
drwxr-xr-x. 2 lebrwcd lebrwcd 6 Feb 11 20:54 Pictures  
drwxr-xr-x. 2 lebrwcd lebrwcd 6 Feb 11 20:54 Public  
drwxr-xr-x. 2 lebrwcd lebrwcd 6 Feb 11 20:54 Templates  
drwxr-xr-x. 2 lebrwcd lebrwcd 6 Feb 11 20:54 Videos  
[lebrwcd@localhost ~]$ cd myredis/  
[lebrwcd@localhost myredis]$ ls      复制redis.conf  
[lebrwcd@localhost myredis]$ cp /etc/redis.conf redis.conf  
[lebrwcd@localhost myredis]$ ls  
redis.conf
```

将/myredis/redis.conf 里的配置项: appendonly 关掉(aof持久化)

之后复制三份:

```
[root@localhost myredis]# touch redis_6379.conf  
[root@localhost myredis]# ls  
redis_6379.conf  redis.conf  
[root@localhost myredis]# vi redis_6379.conf  
[root@localhost myredis]# cp redis_6379.conf redis_6380.conf  
[root@localhost myredis]# cp redis_6379.conf redis_6381.conf  
[root@localhost myredis]# vi redis_6380.conf  
[root@localhost myredis]# vi redis_6381.conf  
[root@localhost myredis]# ll  
total 104  
-rw-r--r--. 1 root      root      94 Mar 21 16:12 redis_6379.conf  
-rw-r--r--. 1 root      root      94 Mar 21 16:13 redis_6380.conf  
-rw-r--r--. 1 root      root      94 Mar 21 16:13 redis_6381.conf  
-rw-r--r--. 1 lebrwcd  lebrwcd  93720 Mar 21 16:10 redis.conf
```

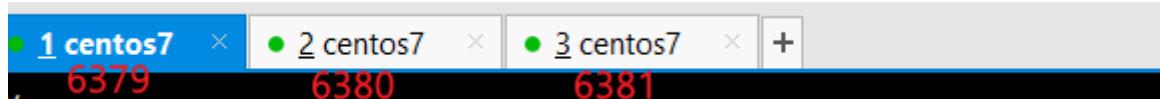
这三个配置文件中的内容为:

```
include /home/lebrwcd/myredis/redis.conf  
pidfile /var/run/redis_6379.pid (6380,6381)  
port 6379 (6380,6381)  
dbfilename dump6379.rdb (6380,6381)
```

之后启动服务:

```
[root@localhost myredis]# redis-server redis_6379.conf
[root@localhost myredis]# ps -ef | grep redis
root      41826      1  0 16:16 ?          00:00:00 redis-server *:6379
root      41832  41582  0 16:16 pts/1    00:00:00 grep --color=auto redis
[root@localhost myredis]# vi redis_6380.conf
[root@localhost myredis]# vi redis_6381.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      41826      1  0 16:16 ?          00:00:00 redis-server *:6379
root      41844      1  0 16:16 ?          00:00:00 redis-server *:6380
root      41850      1  0 16:16 ?          00:00:00 redis-server *:6381
root      41856  41582  0 16:17 pts/1    00:00:00 grep --color=auto redis
```

打开三个窗口模拟一主二从



`redis-cli -p 6379` -p 端口号 指定端口号打开客户端

`info replication` 查看三条主机运行情况

```
[root@localhost myredis]# redis-cli -p 6379
127.0.0.1:6379> info replicaiton
127.0.0.1:6379> info replication
# Replication
role:master 主
connected_slaves:0 从机为0
master_failover_state:no-failover
master_replid:23733bb86c3b3d77cf9783e6c91fbab45cd6f087
master_replid2:0000000000000000000000000000000000000000000000000000000000000000
master_repl_offset:0
second_repl_offset:-1
repl_backlog_active:0
repl_backlog_size:1048576
repl_backlog_first_byte_offset:0
repl backlog_histlen:0
```

没有设置从机的情况下，三条机的运行情况十一样的，我们来设置6380,6381作为6379的从机

在6380,6381客户端下采用命令:`slaveof 主机host 主机端口`

```
127.0.0.1:6380> slaveof 127.0.0.1 6379
OK
127.0.0.1:6380> info replication
# Replication
role:slave 从机
master_host:127.0.0.1
master_port:6379
master_link_status:up
master_last_io_seconds_ago:0
master_sync_in_progress:0
slave_read_repl_offset:28
slave_repl_offset:28
slave_priority:100
slave_read_only:1
replica_announced:1
connected_slaves:0
master_failover_state:no-failover
master_replid:8e63ea17d18112bbe2e4a03bc02e16a2e64984b
master_replid2:0000000000000000000000000000000000000000000000000000000000000000
master_repl_offset:28
second_repl_offset:-1
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:1
repl_backlog_histlen:28
```

再查看主机6379的运行情况

```
127.0.0.1:6379> info replication
# Replication
role:master
connected_slaves:2 两台从机
slave0:ip=127.0.0.1,port=6380,state=online,offset=14,lag=0
slave1:ip=127.0.0.1,port=6381,state=online,offset=14,lag=0
master_failover_state:no-failover
master_replid:8e63ea17d18112bbe2e4a03bc02e16a2e64984b
master_replid2:0000000000000000000000000000000000000000000000000000000000000000
master_repl_offset:14
second_repl_offset:-1
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:1
repl_backlog_histlen:14
```

测试一下主从复制的读写分离：

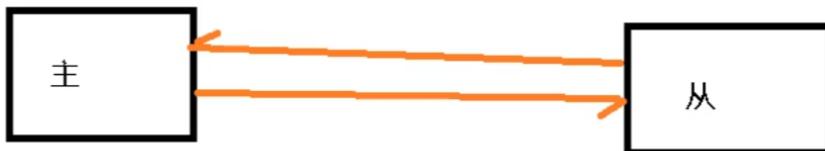
```
127.0.0.1:6379> keys *
(empty array)
127.0.0.1:6379> set k1 v1
OK
127.0.0.1:6379> get k1
"v1"
```

```

127.0.0.1:6380> keys *
1) "k1"
127.0.0.1:6380> get k1
"v1"
从机想要设置k -v，即写操作，发生错误
提示只有读权限
127.0.0.1:6380> set k2 v2
(error) READONLY You can't write against a read only replica.
127.0.0.1:6380>

```

11.1 复制原理



- 1、当从连接上主服务器之后，从服务器 向主服务发送进行数据同步消息
- 2、主服务器接到从服务器发送过来同步消息，把主服务器数据进行持久化，
rdb文件，把rdb文件发送从服务器，从服务器拿到rdb进行读取

- 3、每次主服务器进行写操作之后，和从服务器进行数据同步

11.2 常用三招

11.2.1 一主二仆

- 切入点问题， slave1,slave2从头开始复制还是从切入点开始复制？比如从k4加进来，那之前的k1,k2,k3是否也可以复制？

记住：从机加入进来后，**从头开始复制**主机的数据

- 从机是否可以写？set可否？ 不可以
- 其中一条从机down后，重新加入，还能跟上大部队吗 可以，从机down后，重启会变成主机，如果重新加入进来，会从头开始复制主机
- 主机down后，从机上位还是原地待命？
 - 想象成刘备关羽张飞，刘备生病一段时间，张飞关羽还是不离不弃(info replication)
- 等主机回来，主机新增记录，从机还能否顺利复制？ 可以，大哥回来了，小弟依然听话

```
127.0.0.1:6379> shutdown
not connected>
not connected> exit
[lebrwcd@localhost myredis]$ redis-server redis_6379.conf
[lebrwcd@localhost myredis]$ redis-cli -p 6379
127.0.0.1:6379> keys *
1) "k3"
2) "k1"
3) "k2"
4) "k4"
127.0.0.1:6379> set k5 v5
OK
```

```
127.0.0.1:6380> keys *
```

1) "k2"
2) "k1"
3) "k5"
4) "k4"
5) "k3"

11.2.2 薪火相传

```
master_follower:node-no-follower
master_replid:5fa13468c4e8ad57fc06f1a78bcf40fa4c2f472
master_replid2:0000000000000000000000000000000000000000000000000000000000000000
master_repl_offset:1788
second_repl_offset::1
rep1 backlog_active:1
rep1 backlog_size:1048576
rep1 backlog_first_byte_offset:1
rep1 backlog_histlen:1788
127.0.0.1:6370> info replication
# Replication
role:master
connected_slaves:1
-slave0:ip=127.0.0.1,port=6380,state=online,offset=1928,lag=0
master_follower:node-no-follower
master_replid:5fa13468c4e8ad57fc06f1a78bcf40fa4c2f472
master_replid2:0000000000000000000000000000000000000000000000000000000000000000
master_repl_offset:1928
second_repl_offset::1
rep1 backlog_active:1
rep1 backlog_size:1048576
rep1 backlog_first_byte_offset:1
rep1 backlog_histlen:1830
127.0.0.1:6370> info replication
# Replication
role:slave
connected_slaves:1
slave0:ip=127.0.0.1,port=6380,state=online,offset=1830,lag=0
master_follower:node-no-follower
master_replid:5fa13468c4e8ad57fc06f1a78bcf40fa4c2f472
master_replid2:0000000000000000000000000000000000000000000000000000000000000000
master_repl_offset:1830
second_repl_offset::1
rep1 backlog_active:1
rep1 backlog_size:1048576
rep1 backlog_first_byte_offset:1
rep1 backlog_histlen:1830
127.0.0.1:6380> info replication
# Replication
role:slave
connected_slaves:1
slave0:ip=127.0.0.1,port=6380,state=online,offset=1830,lag=0
master_follower:node-no-follower
master_replid:5fa13468c4e8ad57fc06f1a78bcf40fa4c2f472
master_replid2:0000000000000000000000000000000000000000000000000000000000000000
master_repl_offset:1830
second_repl_offset::1
rep1 backlog_active:1
rep1 backlog_size:1048576
rep1 backlog_first_byte_offset:1
rep1 backlog_histlen:1816
127.0.0.1:6380>
```

11.2.3 反客为主

```
127.0.0.1:6379> info replication
# Replication
role:master
connected_slaves:1
slave0:ip=127.0.0.1,port=6380,state=online,offset=1928,lag=0
master_failover_state:no-failover
master_replid:5fa134d68c4e8a637fc06fla70bcf40fa4c2f472
master_replid2:000000000000000000000000000000000000000000000000000000000000000
master_repl_offset:1928
second_repl_offset:-1
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:1
repl_backlog_histlen:1928
127.0.0.1:6379> shutdown
not connected> exit
[lebrwcd@localhost myredis]$
```



11.3 哨兵模式

可以看作是“反客为主”的自动过程

- 在/myredis 目录中新建 "sentinel.conf" touch sentinel.conf

在文件中输入内容:

```
sentinel monitor mymaster 127.0.0.1 1  
//翻译过来就是：哨兵 监控 主机(哨兵给监控对象起的名，主机ip，主机端口号) 1个哨兵负责
```

- 接着启动哨兵模式：在/usr/local/bin 目录中有这样一个指令：redis-sentinel

```
[lebrwcd@localhost myredis]$ cd /usr/local/bin/  
[lebrwcd@localhost bin]$ ll  
total 18916  
-rw-r--r--. 1 root root 2079 Mar 20 20:24 appendonly.aof  
-rw-r--r--. 1 root root 197 Mar 20 20:35 dump.rdb  
-rwxr-xr-x. 1 root root 4830528 Mar 18 11:24 redis-benchmark  
lrwxrwxrwx. 1 root root 12 Mar 18 11:24 redis-check-aof -> redis-server  
lrwxrwxrwx. 1 root root 12 Mar 18 11:24 redis-check-rdb -> redis-server  
-rwxr-xr-x. 1 root root 5004856 Mar 18 11:24 redis-cli  
lrwxrwxrwx. 1 root root 12 Mar 18 11:24 redis-sentinel -> redis-server  
-rwxr-xr-x. 1 root root 9521760 Mar 18 11:24 redis-server  
[lebrwcd@localhost bin]$
```

- 在/myredis目录输入: redis-sentinel sentinel.conf

```
[lebrwcd@localhost myredis]$ redis-sentinel sentinel.conf
45260:X 21 Mar 2022 20:30:16.886 # o000o000o000o Redis is starti
ng o000o000o000o
45260:X 21 Mar 2022 20:30:16.886 # Redis version=6.2.6, bits=64,
commit=00000000, modified=0, pid=45260, just started
45260:X 21 Mar 2022 20:30:16.886 # Configuration loaded
45260:X 21 Mar 2022 20:30:16.887 # You requested maxclients of 1
0000 requiring at least 10032 max file descriptors.
45260:X 21 Mar 2022 20:30:16.887 # Server can't set maximum open
files to 10032 because of OS error: Operation not permitted.
45260:X 21 Mar 2022 20:30:16.887 # Current maximum open files is
4096. maxclients has been reduced to 4064 to compensate for low
ulimit. If you need higher maxclients increase 'ulimit -n'.
45260:X 21 Mar 2022 20:30:16.887 * monotonic clock: POSIX clock_
gettime
```



```
45260:X 21 Mar 2022 20:30:16.888 # WARNING: The TCP backlog sett
ing of 511 cannot be enforced because /proc/sys/net/core/somaxco
nn is set to the lower value of 128.
45260:X 21 Mar 2022 20:30:16.890 # Sentinel ID is 6f00fe0494947f
43ea80224df61aad2a80669b3b 哨兵监管的主机：其下面有两台从机
45260:X 21 Mar 2022 20:30:16.890 # +monitor master mymaster 127.
0.0.1 6379 quorum 1
45260:X 21 Mar 2022 20:30:16.892 * +slave slave 127.0.0.1:6380 1
27.0.0.1 6380 @ mymaster 127.0.0.1 6379
45260:X 21 Mar 2022 20:30:16.894 * +slave slave 127.0.0.1:6381 1
27.0.0.1 6381 @ mymaster 127.0.0.1 6379
```

- 当我们主机down掉之后，哨兵会根据规则进行从机->主机的转换

```
45260:X 21 Mar 2022 20:32:07.846 # +sdown master mymaster 127.0.0.1 6379
45260:X 21 Mar 2022 20:32:07.846 # +odown master mymaster 127.0.0.1 6379 #quoru
m 1/1
45260:X 21 Mar 2022 20:32:07.846 # +new-epoch 1
45260:X 21 Mar 2022 20:32:07.846 # +try-failover master mymaster 127.0.0.1 6379
45260:X 21 Mar 2022 20:32:07.853 # +vote-for-leader 6f00fe0494947f43ea80224df61
aad2a80669b3b 1 40位的runid
45260:X 21 Mar 2022 20:32:07.853 # +elected-leader master mymaster 127.0.0.1 63
79 故障转移状态 将down掉的主机变为从机
45260:X 21 Mar 2022 20:32:07.853 # +failover-state-select-slave master mymaster
127.0.0.1 6379
45260:X 21 Mar 2022 20:32:07.926 # +selected-slave slave 127.0.0.1:6381 127.0.0
.1 6381 @ mymaster 127.0.0.1 6379 古战转移状态 slaveof no one -> 反客为主
45260:X 21 Mar 2022 20:32:07.926 * +failover-state-send-slave-of-noone slave 127
.0.0.1:6381 127.0.0.1 6381 @ mymaster 127.0.0.1 6379 将6381作为新主机
45260:X 21 Mar 2022 20:32:08.019 * +failover-state-wait-promotion slave 127.0.0
.1:6381 127.0.0.1 6381 @ mymaster 127.0.0.1 6379 等待提拔从机作为新主机...
45260:X 21 Mar 2022 20:32:08.086 # +promoted-slave slave 127.0.0.1:6381 127.0.0
.1 6381 @ mymaster 127.0.0.1 6379
45260:X 21 Mar 2022 20:32:08.086 # +failover-state-reconf-slaves master mymaste
r 127.0.0.1 6379
45260:X 21 Mar 2022 20:32:08.174 * +slave-reconf-sent slave 127.0.0.1:6380 127.
0.0.1 6380 @ mymaster 127.0.0.1 6379
45260:X 21 Mar 2022 20:32:09.094 * +slave-reconf-inprog slave 127.0.0.1:6380 12
7.0.0.1 6380 @ mymaster 127.0.0.1 6379
45260:X 21 Mar 2022 20:32:09.094 * +slave-reconf-done slave 127.0.0.1:6380 127.
0.0.1 6380 @ mymaster 127.0.0.1 6379
45260:X 21 Mar 2022 20:32:09.157 # +failover-end master mymaster 127.0.0.1 6379
45260:X 21 Mar 2022 20:32:09.157 # +switch-master mymaster 127.0.0.1 6379 127.0
.0.1 6381 转移结束
45260:X 21 Mar 2022 20:32:09.157 * +slave slave 127.0.0.1:6380 127.0.0.1 6380 @
mymaster 127.0.0.1 6381
45260:X 21 Mar 2022 20:32:09.157 * +slave slave 127.0.0.1:6379 127.0.0.1 6379 @
mymaster 127.0.0.1 6381
45260:X 21 Mar 2022 20:32:39.210 # +sdown slave 127.0.0.1:6379 127.0.0.1 6379 @
mymaster 127.0.0.1 6381
```

- 重启旧主机，查看三台机的情况 `info replication`

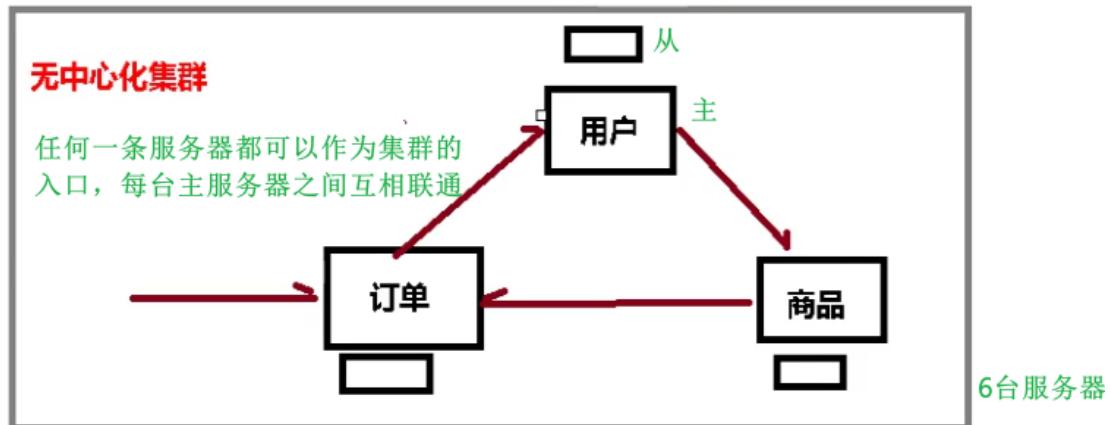
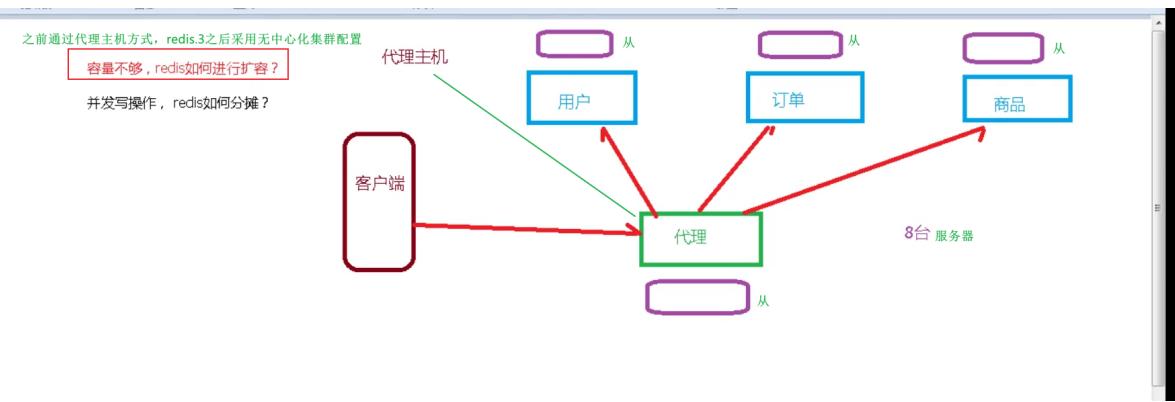
```
[root@node1 ~]# /opt/cockroach bin/cockroach init -n=2  
[root@node1 ~]# /opt/cockroach bin/cockroach start --log-level=info  
[root@node1 ~]# /opt/cockroach bin/cockroach keys add testkey  
[root@node1 ~]# /opt/cockroach bin/cockroach keys get testkey  
[root@node1 ~]# /opt/cockroach bin/cockroach keys list  
[root@node1 ~]# /opt/cockroach bin/cockroach keys delete testkey  
[root@node1 ~]# /opt/cockroach bin/cockroach keys list
```

```
mysql> backlog@127.0.0.1:6307
repl_backlog_first_byte_offset:29
repl_backlog_histlist:14
127.0.0.1:6301> info replication
Error: Broken pipe
127.0.0.1:6301> info replication
# Replication
role:master
connected Slaves:2
slave0_ip:127.0.0.1,port=6300,state=online,of set=25710,lag=0
slave1_ip:127.0.0.1,port=6379,state=online,of set=770,lag=0
master_follower_state:no-follower
master_replid:9652769ad21d7e3ea6df97b85e289ac9dbd3
master_replid:2bf5f6d8886e69c5b157f9d3be455afaa0ff98
master_repl_offset:25710
second_repl_offset:6121
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:29
repl_backlog_histlist:25682
127.0.0.1:6301>
```

十二、集群

1. 容量不够，redis 如何进行扩容？
 2. 并发写操作，redis 如何分摊？

另外，主从模式，薪火相传模式，主机宕机，导致 ip 地址发生变化，应用程序中配置需要修改对应的主机地址、端口等信息。之前通过代理主机来解决，但是 redis3.0 中提供了解决方案。就是**无中心化集群配置**



12.1 什么是集群？

Redis集群实现了对Redis的水平扩容，即启动N个redis节点，将整个数据库分布存储在N个节点中，每个节点存储总数据的1/N(依据slot机制实现)

Redis集群通过**分区(partition)**来提供一定程度的可用性：即使集群中有一部分节点失效或者无法进行通讯，集群也可以继续处理命令请求，因为集群中的每台服务器都是通往集群的入口，每台服务器之间是相互连通的

12.2 模拟集群

12.2.1 删除持久化数据

将/myredis下的rdb, aof文件都删除掉

12.2.2 制作六个实例

模拟六台服务器，端口号为 6379,6380,6381,6389,6390,6391

- 配置基准配置文件redis6479.conf

- 开启 daemonize yes
- Pid 文件名字
- 指定端口
- Log 文件名字

- Dump.rdb 名字
- Appendonly 关掉或者换名字
- 加上关于集群的相关配置
 - 开始集群模式: `cluster-enable yes`
 - 设定节点配置文件名: `cluster-config-file nodes-xxx.conf`
 - 设定节点失联时间。超过改时间(毫秒), 集群自动进行主从切换:
`cluster-node-timeout 毫秒`
- 整个配置文件如下:

```
include /home/lebrwcd/myredis/redis.conf
port 6379
pidfile "/var/run/redis_6379.pid"
dbfilename "dump6379.rdb"
//dir "/home/bigdata/redis_cluster"
//logfile "/home/lebrwcd/myredis/redis_cluster/redis_err_6379.log"
cluster-enabled yes
cluster-config-file nodes-6379.conf
cluster-node-timeout 15000
```

- 修改好redis6379.conf配置文件后, 拷贝5份

redis6379.conf
 redis6380.conf
 redis6381.conf
 redis6389.conf
 redis6390.conf
 redis6391.conf

- 使用查找替换修改另外5个配置文件

`:%s/查找内容/替换内容 :%s/6379/6980` 依此类推

- 启动6个reidis服务

```
[root@localhost myredis]# ps -ef | grep redis
root      54725  54235  0 16:34 pts/1    00:00:00 grep --color=auto redis
[root@localhost myredis]# redis-server redis6379.conf
[root@localhost myredis]# redis-server redis6380.conf
[root@localhost myredis]# redis-server redis6381.conf
[root@localhost myredis]# redis-server redis6389.conf
[root@localhost myredis]# redis-server redis6390.conf
[root@localhost myredis]# redis-server redis6391.conf
[root@localhost myredis]# ps -ef | grep redis
root      54737      1  0 16:35 ?        00:00:00 redis-server *:6379 [cluster]
root      54743      1  0 16:35 ?        00:00:00 redis-server *:6380 [cluster]
root      54749      1  0 16:36 ?        00:00:00 redis-server *:6381 [cluster]
root      54755      1  0 16:36 ?        00:00:00 redis-server *:6389 [cluster]
root      54761      1  0 16:36 ?        00:00:00 redis-server *:6390 [cluster]
root      54767      1  0 16:36 ?        00:00:00 redis-server *:6391 [cluster]
root      54773  54235  0 16:36 pts/1    00:00:00 grep --color=auto redis
```

12.2.3 将六个节点合成一个集群

组合之前，确保所有redis实例启动后，node-xxx.conf文件都生成正常

```
[root@localhost myredis]# ll
total 144
-rw-r--r--. 1 root root 114 Mar 22 16:35 nodes-6379.conf
-rw-r--r--. 1 root root 114 Mar 22 16:35 nodes-6380.conf
-rw-r--r--. 1 root root 114 Mar 22 16:36 nodes-6381.conf
-rw-r--r--. 1 root root 114 Mar 22 16:36 nodes-6389.conf
-rw-r--r--. 1 root root 114 Mar 22 16:36 nodes-6390.conf
-rw-r--r--. 1 root root 114 Mar 22 16:36 nodes-6391.conf
-rw-r--r--. 1 root root 194 Mar 22 16:33 redis6379.conf
-rw-r--r--. 1 root root 194 Mar 22 16:34 redis6380.conf
-rw-r--r--. 1 root root 194 Mar 22 16:34 redis6381.conf
-rw-r--r--. 1 root root 194 Mar 22 16:34 redis6389.conf
-rw-r--r--. 1 root root 194 Mar 22 16:34 redis6390.conf
-rw-r--r--. 1 root root 194 Mar 22 16:34 redis6391.conf
-rw-r--r--. 1 lebrwcd lebrwcd 93720 Mar 22 16:12 redis.conf
-rw-r--r--. 1 lebrwcd lebrwcd 421 Mar 21 20:32 sentinel.conf
```

- 合体： cd 到一开始redis解压的地方， cd /opt/redis-6.2.5/src

下面有个文件

```
root root 363 Mar 18 11:24 redis-cli.d
root root 1107128 Mar 18 11:24 redis-cli.o
root root 66774 Oct 4 18:59 redismodule.h
root root 9521760 Mar 18 11:24 redis-sentinel
root root 9521760 Mar 18 11:24 redis-server
root root 3600 Oct 4 18:59 redis-trib.rb
root root 2591 Mar 18 11:22 release.c
```

可以用它来合体

需要单独安装ruby缓解 (redis5)

```
./redis-trib.rb create --replicas 1 192.168.11.101:6379
192.168.11.101:6380 192.168.11.101:6381 192.168.11.101:6389
192.168.11.101:6390 192.168.11.101:6391
```

也可以采用下面这种方式:(redis6)

```
redis-cli --cluster create --cluster-replicas 1
192.168.11.101:6379 192.168.11.101:6380
192.168.11.101:6381 192.168.11.101:6389
192.168.11.101:6390 192.168.11.101:6391
```

注意：此处不要用 127.0.0.1， 请用真实 IP 地址

--replicas 1 表示采用最简单的方式配置集群，一组：一主一从，正好三组

```

[root@localhost src]# redis-cli --cluster create --cluster-replicas 1 192.168.110.131:6379 1
92.168.110.131:6380 192.168.110.131:6381 192.168.110.131:6389 192.168.110.131:6390 192.168.1
10.131:6391
>>> Performing hash slots allocation on 6 nodes...
Master[0] -> Slots 0 - 5460
Master[1] -> Slots 5461 - 10922 分配插槽，只有master才有
Master[2] -> Slots 10923 - 16383
Adding replica 192.168.110.131:6390 to 192.168.110.131:6379
Adding replica 192.168.110.131:6391 to 192.168.110.131:6380
Adding replica 192.168.110.131:6389 to 192.168.110.131:6381
>>> Trying to optimize slaves allocation for anti-affinity
[WARN] Some slaves are in the same host as their master
M: d018fb5a691ab72b15feeb1ef9a7c242e27655f5 192.168.110.131:6379
  slots:[0-5460] (5461 slots) master
M: dcef37a72e7a63a1d2a885abc707319014827c9a 192.168.110.131:6380
  slots:[5461-10922] (5462 slots) master
M: c1d5f732040a6c294cfda9ae920200e458d9b86c 192.168.110.131:6381
  slots:[10923-16383] (5461 slots) master
S: 673209cd1d8d676b37222fb3f22ffa7d1244523b 192.168.110.131:6389
  replicates c1d5f732040a6c294cfda9ae920200e458d9b86c
S: 47f32f61c6065c69a9cc208611f4e7fb91eee355 192.168.110.131:6390
  replicates d018fb5a691ab72b15feeb1ef9a7c242e27655f5
S: 2adfd7378542f541aaaf703dcc193f30cd52e43ab 192.168.110.131:6391
  replicates dcef37a72e7a63a1d2a885abc707319014827c9a
Can I set the above configuration? (type 'yes' to accept): yes
>>> Nodes configuration updated
>>> Assign a different config epoch to each node
>>> Sending CLUSTER MEET messages to join the cluster
Waiting for the cluster to join 开始集群

```

Waiting for the cluster to join

```

..
>>> Performing Cluster Check (using node 192.168.110.131:6379)
M: d018fb5a691ab72b15feeb1ef9a7c242e27655f5 192.168.110.131:6379
  slots:[0-5460] (5461 slots) master
  1 additional replica(s)
S: 2adfd7378542f541aaaf703dcc193f30cd52e43ab 192.168.110.131:6391
  slots: (0 slots) slave
  replicates dcef37a72e7a63a1d2a885abc707319014827c9a
S: 47f32f61c6065c69a9cc208611f4e7fb91eee355 192.168.110.131:6390
  slots: (0 slots) slave
  replicates d018fb5a691ab72b15feeb1ef9a7c242e27655f5
S: 673209cd1d8d676b37222fb3f22ffa7d1244523b 192.168.110.131:6389
  slots: (0 slots) slave
  replicates c1d5f732040a6c294cfda9ae920200e458d9b86c
M: c1d5f732040a6c294cfda9ae920200e458d9b86c 192.168.110.131:6381
  slots:[10923-16383] (5461 slots) master
  1 additional replica(s)
M: dcef37a72e7a63a1d2a885abc707319014827c9a 192.168.110.131:6380
  slots:[5461-10922] (5462 slots) master
  1 additional replica(s)
[OK] All nodes agree about slots configuration. 所有节点同意插槽的配置
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered. 插槽共有16384个【0~16383】

```

12.2.4 集群模式下登录客户端

- 普通方式 `redis-cli -p 端口号`

可能直接进入读主机，存储数据时，会出现MOVED重定向操作，所以应该以集群方式登录

```

[root@zy src]# redis-cli -p 6379
127.0.0.1:6379> keys *
(empty list or set)
127.0.0.1:6379> set k1 v1
(error) MOVED 12706 192.168.137.3:6381

```

- 集群策略连接 `redis-cli -c -p 端口号`

采用集群策略连接，设置数据后根据key-slot规则会自动重定向到相应的写主机

```
[root@localhost src]# redis-cli -c -p 6379
127.0.0.1:6379> keys *
(empty array)
127.0.0.1:6379> set k1 v1
-> Redirected to slot [12706] located at 192.168.110.131:6381
OK
192.168.110.131:6381> keys *
1) "k1"
```

- 查看集群信息

通过 `cluster nodes` 命令可以查看集群的信息

```
192.168.110.131:6381> cluster nodes
c1d5f732040a6c294cfda9ae920200e458d9b86c 192.168.110.131:6381@16381 myself,master - 0 164793
8843000 3 connected 10923-16383
673209cd1d8d676b37222fb3f22ffa7d1244523b 192.168.110.131:6389@16389 slave c1d5f732040a6c294c
fda9ae920200e458d9b86c 0 1647938847000 3 connected
2adfd7378542f541aaaf703dcc193f30cd52e43ab 192.168.110.131:6391@16391 slave dcef37a72e7a63a1d2
a885abc707319014827c9a 0 1647938848185 2 connected
47f32f61c6065c69a9cc208611f4e7fb91eee355 192.168.110.131:6390@16390 slave d018fb5a691ab72b15
feeb1ef9a7c242e27655f5 0 1647938847000 1 connected
dcef37a72e7a63a1d2a885abc707319014827c9a 192.168.110.131:6380@16380 master - 0 1647938845000
2 connected 5461-10922
d018fb5a691ab72b15feeb1ef9a7c242e27655f5 192.168.110.131:6379@16379 master - 0 1647938847159
1 connected 0-5460
```

12.2.5 redis cluster 如何分配节点

一个集群至少要有三个主节点

1. 选项 `--cluster-replicas 1` 表示我们希望为集群中的每个主节点创建一个从节点。
2. 分配原则尽量保证每个主数据库运行在不同的 IP 地址，每个从库和主库不在一个 IP 地址上。

12.3 slots 插槽

```
[OK] All nodes agree about slot coverage...
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.
```

- 一个 Redis 集群包含 16384 个插槽 (hash slot)， 数据库中的每个键都属于这 16384 个插槽的其中一个
- 集群使用公式 $\text{CRC16}(\text{key}) \% 16384$ 来计算键 key 属于哪个槽， 其中 `CRC16(key)` 语句用于计算键 key 的 CRC16 校验和

```
1) <connected>
192.168.110.131:6381> hset user:1001 name wcd age 18 sex male
-> Redirected to slot [5712] located at 192.168.110.131:6380
(integer) 3
192.168.110.131:6380> keys *
1) "user:1001"
192.168.110.131:6380> get user:1001
```

- 集群中的每个节点负责处理一部分插槽。举个例子，如果一个集群可以有主节点，其中：
 - 节点 A 负责处理 0 号至 5460 号插槽。
 - 节点 B 负责处理 5461 号至 10922 号插槽。
 - 节点 C 负责处理 10923 号至 16383 号插槽。

12.4 在集群中录入值

在 redis-cli 每次录入、查询键值，redis 都会计算出该 key 应该送往的插槽，如果不是该客户端对应服务器的插槽，redis 会报错，并告知应前往的 redis 实例地址和端口。

- redis-cli 客户端提供了 `-c` 参数实现自动重定向。
- 如 redis-cli -c -p 6379 登录后，再录入、查询键值对可以自动重定向。
- 不在一个 slot 下的键值，是不能使用 mget,mset 等多键操作。

```
[root@localhost src]# redis-cli -c -p 6379
127.0.0.1:6379> keys *
(empty array)
o 127.0.0.1:6379> set k1 v1
-> Redirected to slot [12706] located at 192.168.110.131:6381
OK
192.168.110.131:6381> keys *
1) "k1"
127.0.0.1:6381> set k2 v3
-> Redirected to slot [449] located at 192.168.110.131:6379
OK
o 192.168.110.131:6379> keys *
1) "k2"
192.168.110.131:6379> mget k1 k2
(error) CROSSLINK Keys in request don't hash to the same slot
```

k1和k2是不同slot下的键值，不能使用mget操作

- 可以通过{}来定义组的概念，从而使key中{}内相同内容的键值对放到同一个slot 中去

简单的说就是 key{组名} value ,组名一样的被放到同一个slot中，之后就可以使用mset,mget

```
192.168.110.131:6379> mset k1 v11 k2 v22 之前k1, k2不在同一个slot, 不能mset
(error) CROSSLINK Keys in request don't hash to the same slot
192.168.110.131:6379> mset k1{kg} v11 k2{kg} v22
OK
```

```
192.168.110.131:6379> mset k3{kg} v3 k4{kg} v4 k5{kg} v5
OK
192.168.110.131:6379> keys *
1) "k4{kg}"
2) "k3{kg}"
3) "k5{kg}"
4) "k2"
```

12.6 查询集群中的值

`cluster keyslot <组名>` 查看组名为xx的slot

`cluster countkeysinslot <slot>` 查看slot中有多少个键值对

`cluster getkeyinslot <slot> <count>` 返回 count 个 slot 槽中的键

```
192.168.110.131:6379> cluster keyslot kg
(integer) 2961
192.168.110.131:6379> cluster countkeysinslot 2961
(integer) 5
```

```
192.168.110.131:6379> cluster getkeysinslot 2961 5
1) "k1{kg}"
2) "k2{kg}"
3) "k3{kg}"
4) "k4{kg}"
5) "k5{kg}"
192.168.110.131:6379> cluster getkeysinslot 2961 4
1) "k1{kg}"
2) "k2{kg}"
3) "k3{kg}"
4) "k4{kg}"
```

12.7 故障修复

- 如果主节点下线? 从节点能否自动升为主节点? **注意:** 15 秒超时

如果主节点下线了, 根据配置文件中设置的超时时间, 如果超过了这个时间主节点还未重新上线, 将会将它之前的从机转为主机

```
192.168.110.131:6379> shutdown
not connected>
not connected> exit
[root@localhost src]# redis-cli -c -p 6380
127.0.0.1:6380> cluster nodes
2adfd7378542f541aaaf703dcc193f30cd52e43ab 192.168.110.131:6391 slave dcef37a72e7a63a1d2
a885abc707319014827c9a 0 1647939525355 2 connected
673209cd1d8d676b37222fb3f22ffa7d1244523b 192.168.110.131:6389 slave c1d5f732040a6c294c
fda9ae920200e458d9b86c 0 1647939524331 3 connected
c1d5f732040a6c294cfda9ae920200e458d9b86c 192.168.110.131:6381 master - 0 1647939525000
3 connected 10923-16383
dcef37a72e7a63a1d2a885abc707319014827c9a 192.168.110.131:6380 myself,master - 0 164793
9524000 2 connected 5461-10922 超时时间之前还是master
d018fb5a691ab72b15feeb1ef9a7c242e27655f5 192.168.110.131:6379@16379 master - 1647939513155 1
647939505570 1 disconnected 0-5460
47f32f61c6065c69a9cc208611f4e7fb91eee355 192.168.110.131:6390@16390 slave d018fb5a691ab72b15
feeb1ef9a7c242e27655f5 0 1647939526409 1 connected 过于超时时间
127.0.0.1:6380> cluster nodes
2adfd7378542f541aaaf703dcc193f30cd52e43ab 192.168.110.131:6391 slave dcef37a72e7a63a1d2
a885abc707319014827c9a 0 1647939556161 2 connected
673209cd1d8d676b37222fb3f22ffa7d1244523b 192.168.110.131:6389 slave c1d5f732040a6c294c
fda9ae920200e458d9b86c 0 1647939555000 3 connected
c1d5f732040a6c294cfda9ae920200e458d9b86c 192.168.110.131:6381 master - 0 1647939555000
3 connected 10923-16383
dcef37a72e7a63a1d2a885abc707319014827c9a 192.168.110.131:6380 myself,master - 0 164793
9553000 2 connected 5461-10922
d018fb5a691ab72b15feeb1ef9a7c242e27655f5 192.168.110.131:6379@16379 master,fail - 1647939513
155 1647939505570 1 disconnected
47f32f61c6065c69a9cc208611f4e7fb91eee355 192.168.110.131:6390@16390 master - 0 1647939555091
7 connected 0-5460
127.0.0.1:6380> cluster getkeysinslot 2961 5
```

- 主节点恢复后, 主从关系会如何?

主节点回来变成从机。

```
[root@localhost myredis]# redis-cli -c -p 6379
127.0.0.1:6379> cluster nodes
2adfd7378542f541aaf703dcc193f30cd52e43ab 192.168.110.131:6391@16391 slave dcef37a72e7a63a1d2a885abc707319014
827c9a 0 1647947347264 2 connected
dcef37a72e7a63a1d2a885abc707319014827c9a 192.168.110.131:6380@16380 master - 0 1647947346000 2 connected 546
1-10922
c1d5f732040a6c294cfda9ae920200e458d0b86c 192.168.110.131:6381@16381 master - 0 1647947345214 3 connected 109
23-16383
47f32f61c6065c69a9cc208611f4e7fb91eee355 192.168.110.131:6390@16390 master - 0 1647947346000 7 connected 0-5
460
d018fb5a691ab72b15feeb1ef9a7c242e27655f5 192.168.110.131:6379@16379 myself,slave 47f32f61c6065c69a9cc208611f
4e7fb91eee355 0 1647947341000 7 connected
673209cd1d8d676b37222fb3f22ffa7d1244523b 192.168.110.131:6389@16389 slave c1d5f732040a6c294cfda9ae920200e458
d9b86c 0 1647947346240 3 connected
127.0.0.1:6379>
```

- 如果所有某一段插槽的主从节点都宕掉，redis 服务是否还能继续？

取决于 redis.conf 中的参数 `cluster-require-full-coverage`

- 如果某一段插槽的主从都挂掉，而 `cluster-require-full-coverage` 为 **yes**，那么，**整个集群都挂掉**
- 如果某一段插槽的主从都挂掉，而 `cluster-require-full-coverage` 为 **no**，那么，**该插槽数据全都不能使用，也无法存储。**

12.8 Jedis集群开发

即使连接的不是主机，集群会自动切换主机存储。**主机写，从机读。** 无中心化主从集群：无论从哪台主机写的数据，其他主机上都能读到数据。

```
public class JedisClusterTest {
    public static void main(String[] args) {
        Set<HostAndPort> set = new HashSet<HostAndPort>();
        set.add(new HostAndPort("192.168.31.211", 6379));
        JedisCluster jedisCluster = new JedisCluster(set);
        jedisCluster.set("k1", "v1");
        System.out.println(jedisCluster.get("k1"));
    }
}
```

12.9 Redis集群好处

- 实现扩容
- 分摊压力
- 无中心配置相对简单

12.10 Redis集群的不足

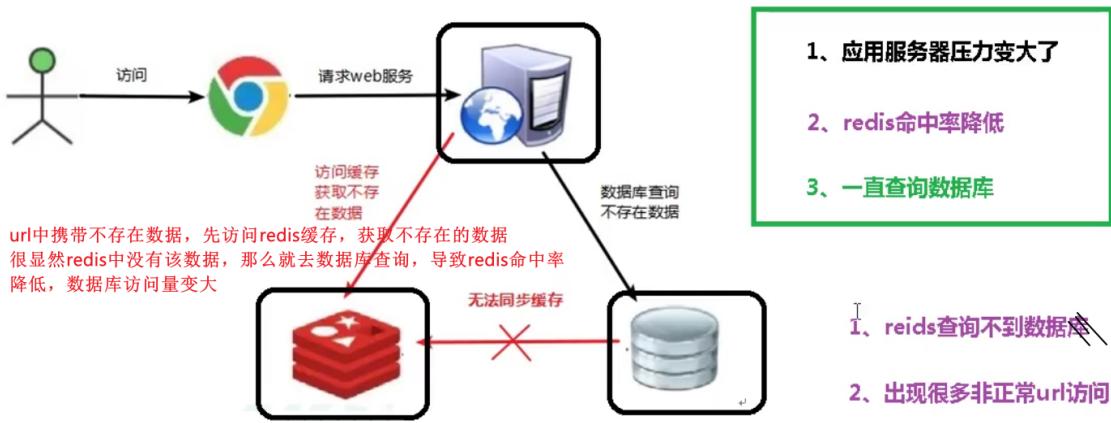
- 多键操作是不被支持的
- 多键的Redis事务时不被支持的，lua脚步不被支持
- 由于集群方案出现较晚，很多公司已经采用了其他的集群方案，而代理或者客户端分片的方案想要迁移至 redis cluster，需要整体迁移而不是逐步过渡，复杂度较大

十三、Redis应用解决问题

13.1 缓存穿透

13.1.1 问题描述

key 对应的数据在数据源并不存在，每次针对此 key 的请求从缓存获取不到，请求都会压到数据源，从而可能压垮数据源。比如用一个不存在的用户 id 获取用户信息，不论缓存还是数据库都没有，若黑客利用此漏洞进行攻击可能压垮数据库。



13.1.2 解决方案

一个一定不存在缓存及查询不到的数据，由于**缓存是不命中时被动写的**，并且出于容错考虑，如果从存储层查不到数据则不写入缓存，这将导致这个不存在的数据每次请求都要到存储层去查询，失去了缓存的意义。

解决方案：

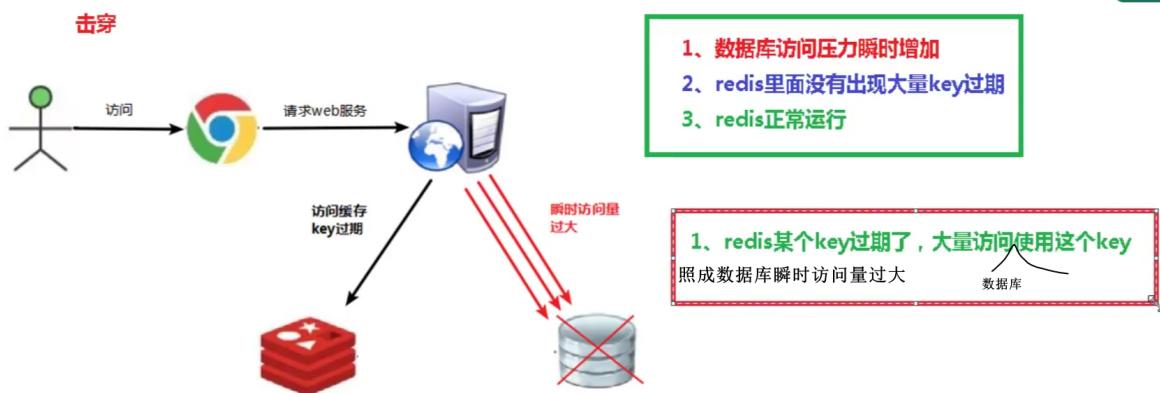
1. 对空值缓存：如果一个查询返回的数据为空（不管是数据是否存在），我们仍然把这个空结果（null）进行缓存，设置空结果的过期时间会很短，最长不超过五分钟
2. 设置可访问的名单（白名单）：使用 bitmaps 类型定义一个可以访问的名单，名单 id 作为 bitmaps 的偏移量，每次访问和 bitmap 里面的 id 进行比较，如果访问 id 不在 bitmaps 里面，进行拦截，不允许访问。
3. 采用布隆过滤器：(布隆过滤器 (Bloom Filter) 是 1970 年由布隆提出的。它实际上是一个很长的二进制向量(位图)和一系列随机映射函数 (哈希函数)。布隆过滤器可以用于检索一个元素是否在一个集合中。它的优点是空间效率和查询时间都远远超过一般的算法，缺点是有一定的误识别率和删除困难。) 将所有可能存在的数据哈希到一个足够大的 bitmaps 中，一个一定不存在的数据会被这个 bitmaps 拦截掉，从而避免了对底层存储系统的查询压力。

4. 进行实时监控：当发现 Redis 的命中率开始急速降低，需要排查访问对象和访问的数据，和运维人员配合，可以设置黑名单限制服务

13.2 缓存击穿

13.2.1 问题概述

key 对应的数据存在，但在 redis 中过期，此时若有大量并发请求过来，这些请求发现缓存过期一般都会从后端 DB 加载数据并回设到缓存，这个时候大并发的请求可能 会瞬间把后端 DB 压垮。

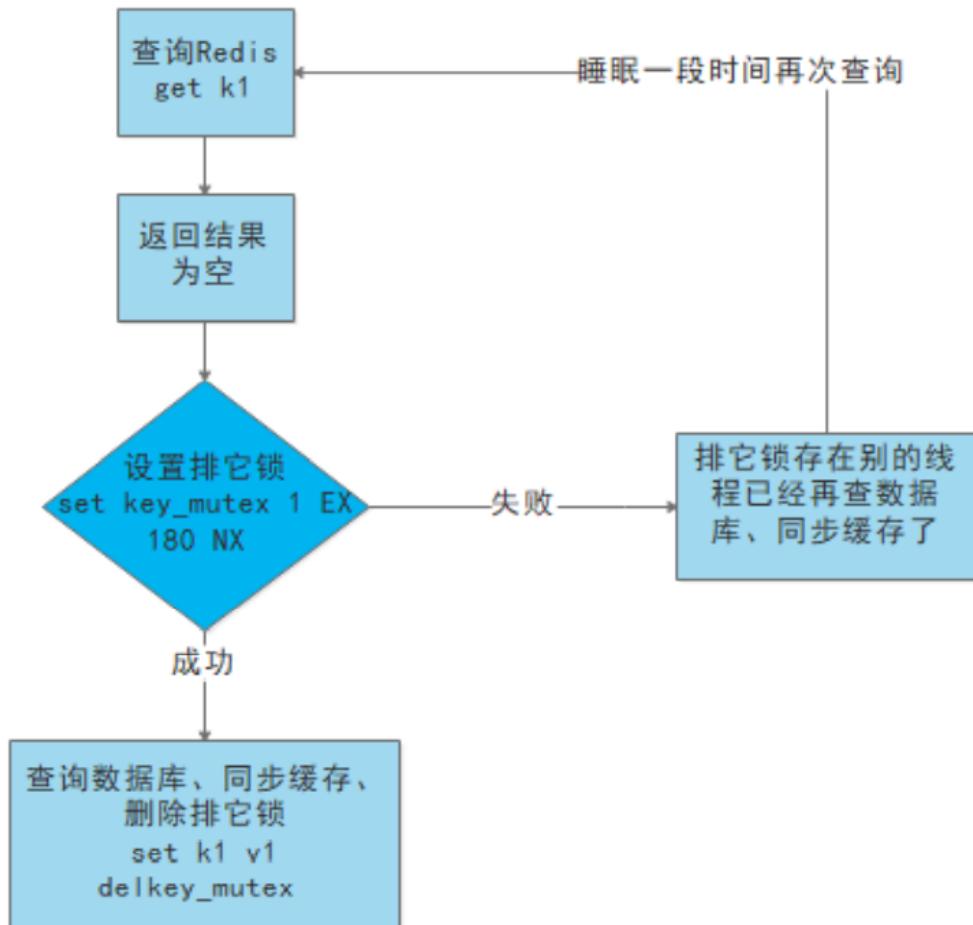


13.2.2 解决方案

key 可能在某些时间点被超高并发地访问，是一种非常“热点”的数据。这个时候，需要考虑一个问题：缓存被“击穿”的问题。

解决方案：

- 预先设置热门数据：在 redis 高峰访问之前，把一些热门数据提前存入到 redis 里面，加大这些热门数据 key 的时长防止过期
- 实时调整：现场监控哪些数据热门，实时调整 key 的过期时长
- 使用锁：
 - 就是在缓存失效的时候（判断拿出来的值为空），不是立即去 load db。（
 - 先使用缓存工具的某些带成功操作返回值的操作（比如 Redis 的 SETNX）去 set 一个 mutex key
 - 当操作返回成功时，再进行 load db 的操作，并回设缓存，最后删除 mutex key；
 - 当操作返回失败，证明有线程在 load db，当前线程睡眠一段时间再重试整个 get 缓存的方法。

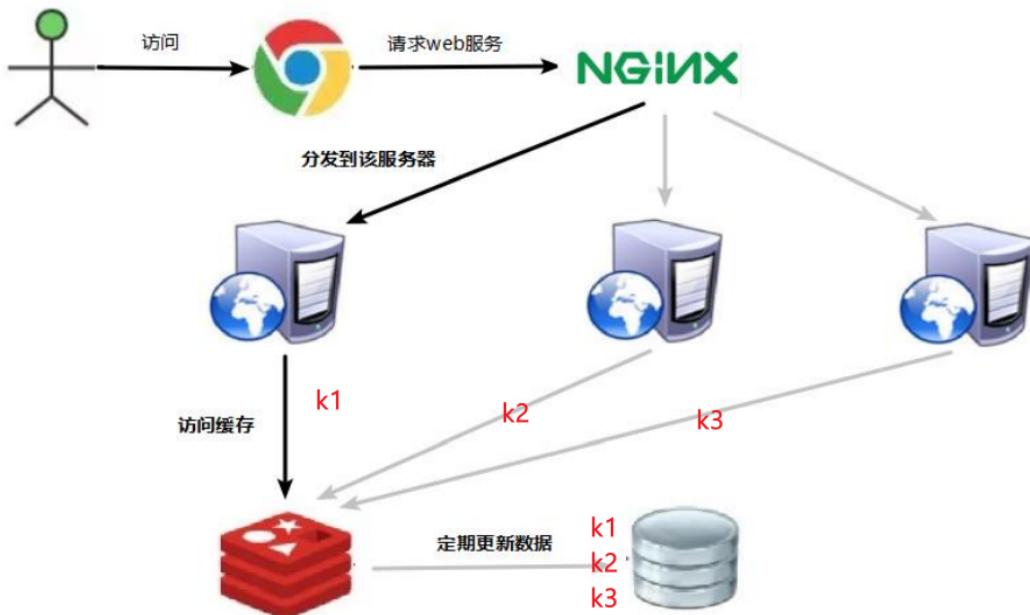


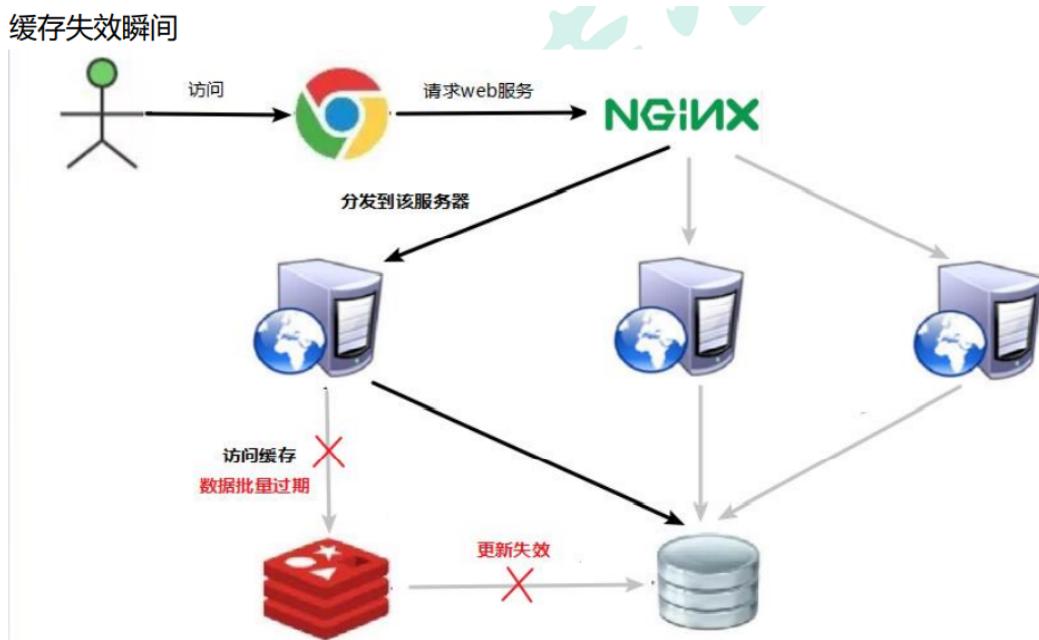
13.3 缓存雪崩

13.3.1 问题描述

大量key 对应的数据存在，但在 redis 中过期，此时若有大量并发请求过来，这些请求发现缓存过期一般都会从后端 DB 加载数据并回设到缓存，这个时候大并发的请求可能会瞬间把后端 DB 压垮。缓存雪崩与缓存击穿的区别在于这里针对很多 key 缓存，前者则是某一个 key

正常访问





13.3.2 解决方案

缓存失效时的雪崩效应对底层系统的冲击非常可怕！

解决方案：

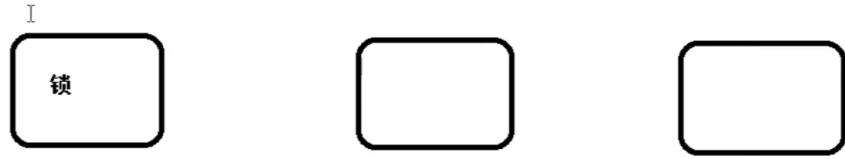
1. 构建多级缓存架构：nginx 缓存 + redis 缓存 + 其他缓存（ehcache 等）
2. 使用锁或队列：用加锁或者队列的方式保证来保证不会有大量的线程对数据库一次性进行读写，从而避免失效时大量的并发请求落到底层存储系统上。不适用高并发情况
3. 设置过期标志更新缓存：记录缓存数据是否过期（设置提前量），如果过期会触发通知另外的线程在后台去更新实际 key 的缓存。
4. 将缓存失效时间分散开：比如我们可以在原有的失效时间基础上增加一个随机值，比如 1-5 分钟随机，这样每一个缓存的过期时间的重复率就会降低，就很难引发集体失效的事件。

13.4 分布式锁

13.4.1 问题描述

随着业务发展的需要，原单体单机部署的系统被演化成分布式集群系统后，由于**分布式系统多线程、多进程并且分布在不同机器上**，这将使原单机部署情况下的并发控制锁策略失效，单纯的 Java API 并不能提供分布式锁的能力。为了解决这个问题就需要一种**跨 JVM 的互斥机制来控制共享资源的访问**，这就是**分布式锁**要解决的问题！

分布式锁 加一把锁，让里面所有的机器都共享这把锁



分布式锁主流的实现方案：

1. 基于数据库实现分布式锁
2. 基于缓存（Redis 等）
3. 基于 Zookeeper

每一种分布式锁解决方案都有各自的优缺点：

1. 性能：redis 最高
2. 可靠性：zookeeper 最高

这里，我们就基于 redis 实现分布式锁。

13.4.2 使用redis实现分布式锁

redis:命令 `set <key> <value> NX/PX/EX millisecond/second`

- EX second : 设置键的过期时间为 second 秒。

SET EX 效果等同于 SETEX 。

```
127.0.0.1:6379> set k1 v1 ex 10
OK
127.0.0.1:6379> ttl k1
(integer) -2
127.0.0.1:6379> setex k1 v1 10 setex <key> <second> <value>
(error) ERR value is not an integer or out of range
127.0.0.1:6379> setex k1 10 v1
OK
127.0.0.1:6379>
```

- PX millisecond : 设置键的过期时间为 millisecond 毫秒。

SET PX 效果等同于 PSETEX 。

```
127.0.0.1:6379> set k2 v2 px 10000
OK
127.0.0.1:6379> ttl k2
(integer) 4
127.0.0.1:6379> ttl k2
(integer) -2
127.0.0.1:6379> psetex k2 10000 v2 psetex <key> <millsec> <value>
OK
127.0.0.1:6379>
```

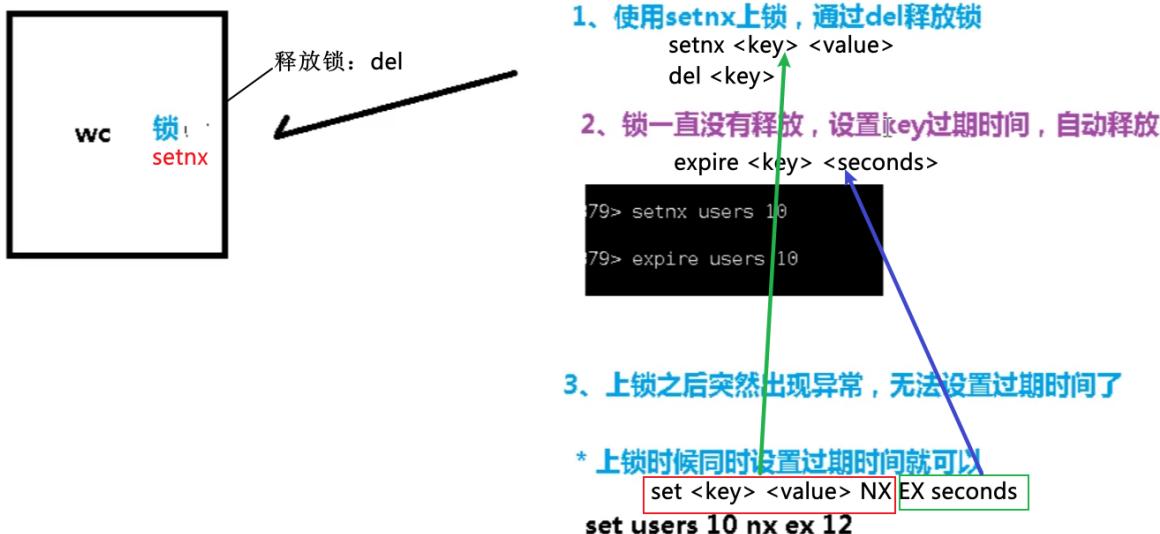
- NX : 只在键不存在时，才对键进行设置操作。

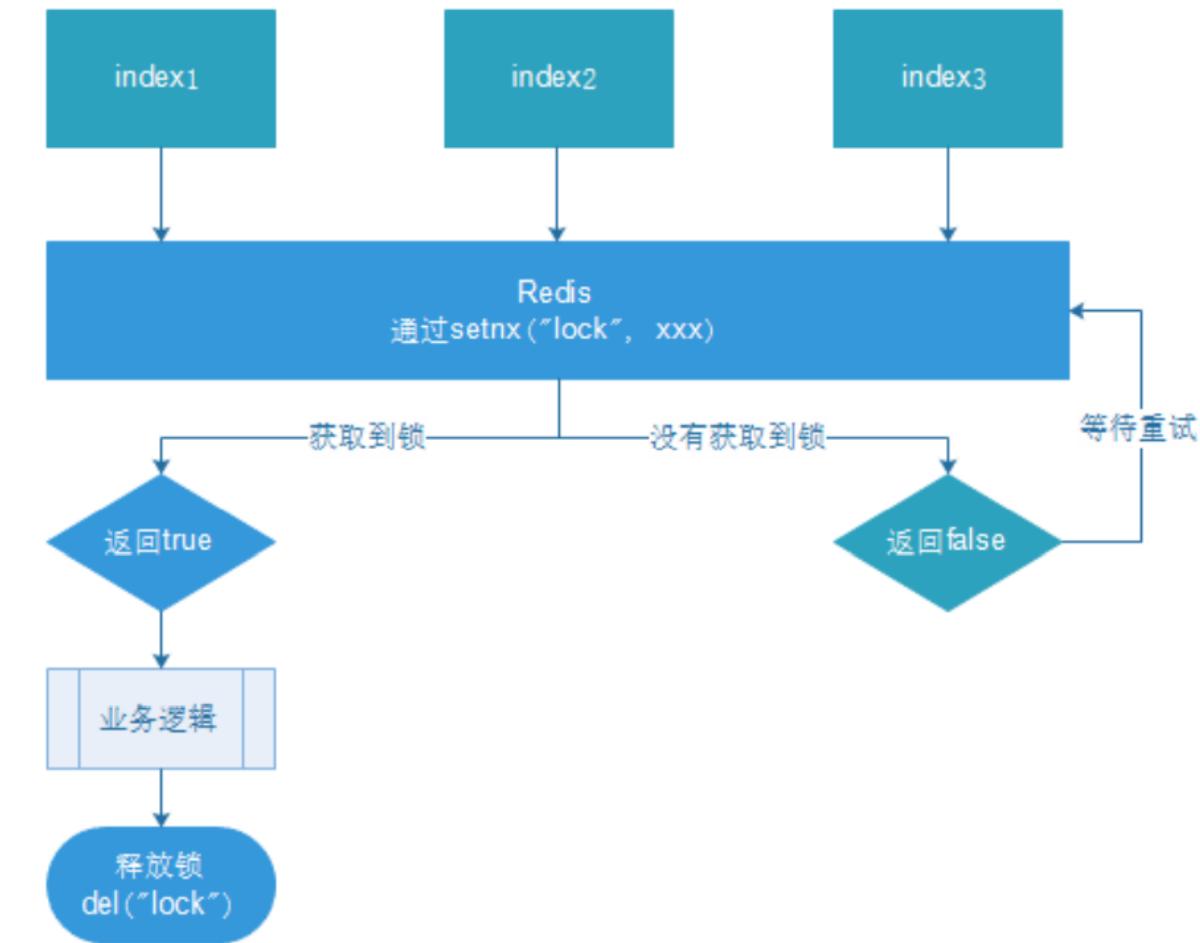
SET NX 效果等同于 SETNX 。

```
127.0.0.1:6379> set k3 v3 nx
OK
127.0.0.1:6379> set k3 v33 nx k3已经存在，无法重新进行
(nil)          设置操作
127.0.0.1:6379> del k3
(integer) 1
127.0.0.1:6379> setnx k3 v3
(integer) 1
127.0.0.1:6379> setnx k3 v33
(integer) 0
127.0.0.1:6379> █
```

- XX：只在键已经存在时，才对键进行设置操作。

```
127.0.0.1:6379> set k4 v4
OK
127.0.0.1:6379> set k4 v44
OK
127.0.0.1:6379> get k4
"v44"
127.0.0.1:6379> █
```





1. 多个客户端同时获取锁 (setnx)
2. 获取成功，执行业务逻辑{从 db 获取数据，放入缓存}，执行完成释放锁 (del)
3. 其他客户端等待重试

13.4.3 代码实现

13.4.3.1 Springboot连接redis

```

<!--
https://mvnrepository.com/artifact/org.springframework.boot/spring-
boot-starter-data-redis -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
    <version>2.6.2</version>
</dependency>

<!-- https://mvnrepository.com/artifact/redis.clients/jedis --
>
<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
    <version>2.9.0</version>
</dependency>

```

13.4.3.2 配置文件

```

spring:
  redis:
    jedis:
      pool:
        enabled: true
      port: 6379
      host: 192.168.110.131
      database: 0
      connect-timeout: 1800000
    lettuce:
      pool:
        max-active: 20
        max-wait: -1
        max-idle: 5
        min-idle: 0

```

13.4.3.3 java代码

```

@RestController
public class uuidController {

    @Autowired
    StringRedisTemplate redisTemplate;

    @GetMapping("testlock")
    public void testlock(){

        //setnx lock 111 上锁
        Boolean lock =
        redisTemplate.opsForValue().setIfAbsent("lock","111");
    }
}

```

```
//获取锁
if(lock){
    String value = redisTemplate.opsForValue().get("num");
    if(StringUtils.isEmpty(value)){
        //如果没有获取到值，直接中止方法
        return;
    }

    //如果获取到值了
    int num = Integer.parseInt(value);
    //把redis的num+1

    redisTemplate.opsForValue().set("num",String.valueOf(++num));
    //释放锁 del lock
    redisTemplate.delete("lock");
}

}else{
    //如果获取不到锁，每隔0.1秒再获取
    try {
        Thread.sleep(100);
        testlock();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}
```

13.4.3.4 ab(网关压力)测试

```
ab -n 1000 -c 100 http://本机IP:8080/testlock
```

```
[lebrwcd@localhost myredis]$ ab -n 1000 -c 100 http://192.168.110.1:8080/testlock
This is ApacheBench, Version 2.3 <$Revision: 1430300 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking 192.168.110.1 (be patient)
Completed 100 requests
Completed 200 requests
Completed 300 requests
Completed 400 requests
Completed 500 requests
Completed 600 requests
Completed 700 requests
Completed 800 requests
Completed 900 requests
Completed 1000 requests
Finished 1000 requests

Server Software:
Server Hostname:          192.168.110.1
Server Port:               8080

Document Path:             /testlock
Document Length:           0 bytes

Write errors:               0
Total transferred:          92000 bytes
HTML transferred:           0 bytes
Requests per second:        102.33 [#/sec] (mean)
Time per request:           977.186 [ms] (mean)
Time per request:           9.772 [ms] (mean, across all concurrent requests)
Transfer rate:              9.19 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:       0    1  2.2      0     11
Processing:    2   586 1487.7     4    8846
Waiting:       1   586 1487.7     4    8846
Total:         2   587 1489.3     4    8853

Percentage of the requests served within a certain time (ms)
 50%      4
 66%     18
 75%    183
 80%    404
 90%   2219
 95%  4443
 98% 6111
 99% 7316
100% 8853 (longest request)
```

查看redis中num 的值

```
[lebrwcd@localhost ~]$ redis-cli
127.0.0.1:6379> get num
"1000"
```

1000次请求， num最后的值是1000， 正确

- **问题：**setnx 刚好获取到锁，业务逻辑出现异常，导致锁无法释放

解决：设置过期时间，自动释放锁。

13.4.4 优化之设置过期时间

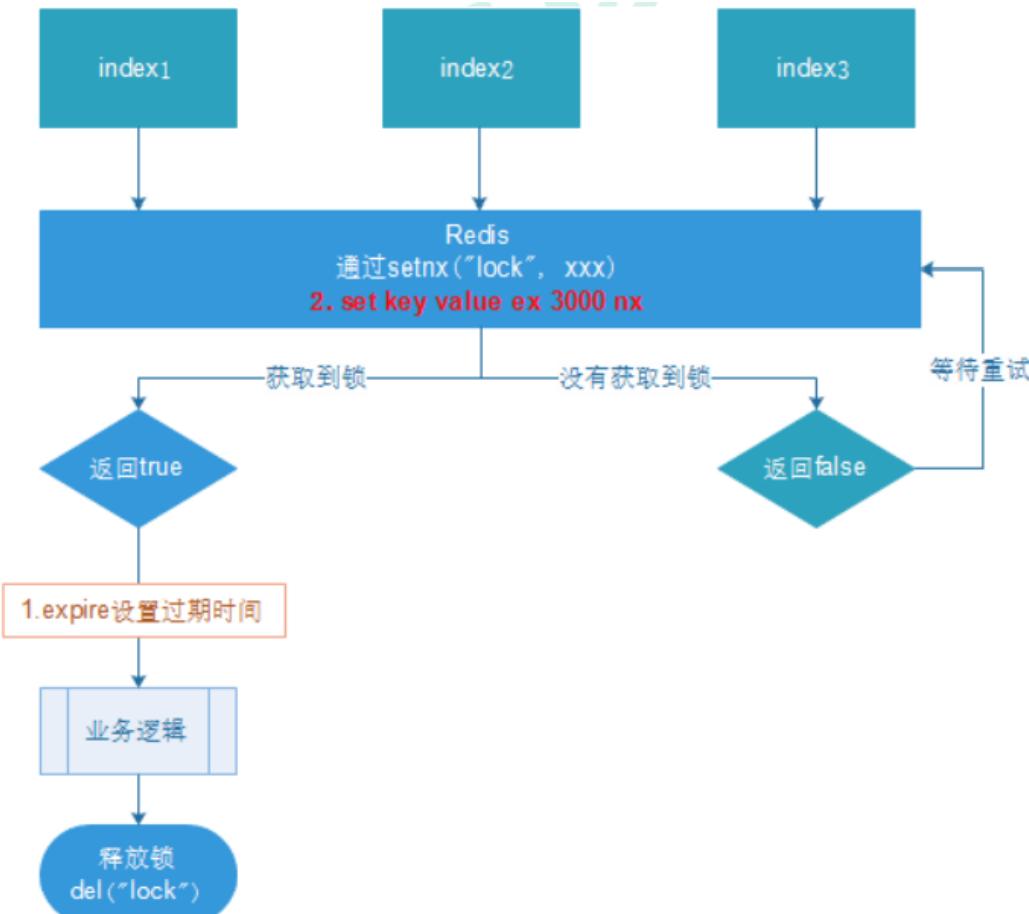
设置过期时间有两种方式：

- 首先想到通过 expire 设置过期时间（缺乏原子性：如果在 setnx 和 expire 之间出现异常，锁也无法释放）

setnx k v + expire k 10

缺乏原子性，如果在设置过期时间的时候，服务器突然断了，那么最后重启服务器，过期时间还是没能设置上

- 在 set 时指定过期时间（推荐）



```
Boolean lock = redisTemplate.opsForValue().setIfAbsent("lock", "111", 3, TimeUnit.SECONDS);
```

压力测试肯定也没有问题。自行测试

- 问题：可能会释放其他服务器的锁。

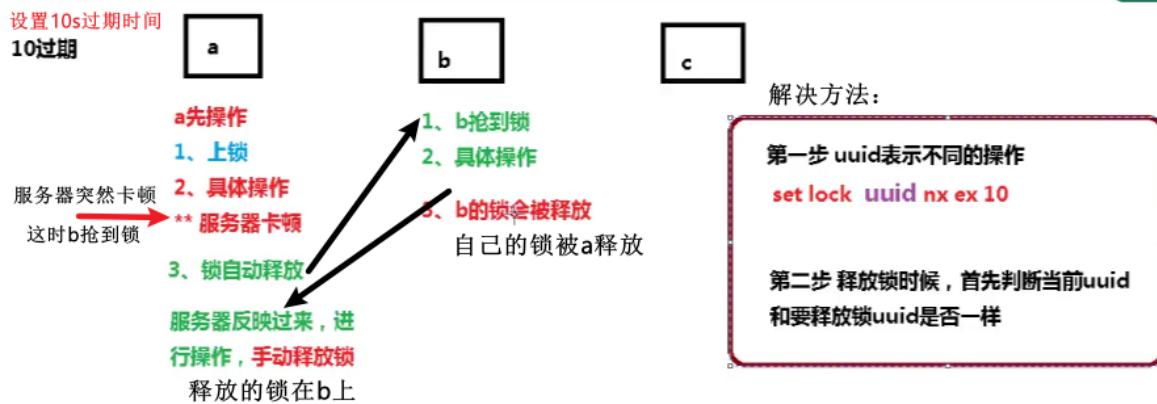
场景：假设index1业务逻辑的执行时间是7s，执行流程如下：

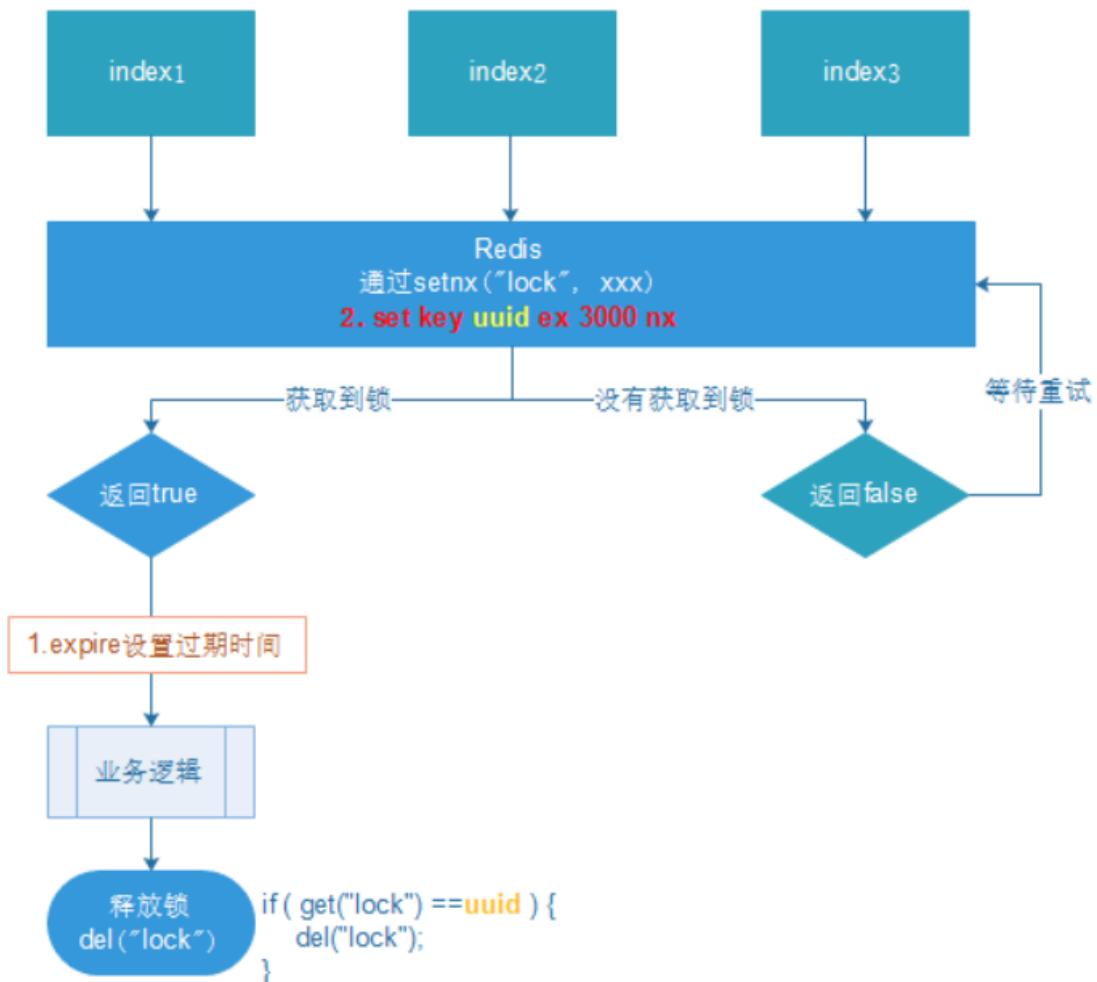


1. index1 业务逻辑没执行完，3秒后锁被自动释放。
 2. index2 获取到锁，执行业务逻辑，3秒后锁被自动释放。
 3. index3 获取到锁，执行业务逻辑
 4. index1 业务逻辑执行完成，开始调用 del 释放锁，这时释放的是 index3 的锁，导致 index3 的业务只执行 1s 就被别人释放。
- 最终等于没锁的情况。

解决：setnx 获取锁时，设置一个指定的唯一值（例如：uuid）；释放前获取这个值，判断是否是自己的锁

13.4.5 优化之UUID防误删





```

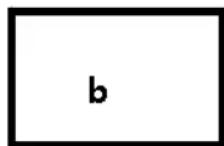
Boolean lock =
redisTemplate.opsForValue().setIfAbsent("lock",uuid,10,
TimeUnit.SECONDS);
...
//释放锁 del lock
if(redisTemplate.opsForValue().get("lock").equals(uuid)){
    redisTemplate.delete("lock");
}

```

问题：删除操作缺乏原子性

原子性操作造成问题

解决方法



1、上锁

2、具体操作

3、释放锁 del

(1) 比较uuid，一样

(2) 删除操作时候，正
要删除，还没有删除，锁
到了过期时间，自动释放

3、删除操作

1、b锁

2、具体操作

3、a释放b的锁

场景：

1. index1 执行删除时，查询到的 lock 值确实和 uuid 相等 uuid=v1
set(lock,uuid);

```
if(redisTemplate.opsForValue().get("lock").equals(uuid)){
```

2. index1 执行删除前，lock 刚好过期时间已到，被 redis 自动释放。在 redis 中没有了 lock，没有了锁。

```
redisTemplate.delete( key: "lock");
```

3. index2 获取了 lock。index2 线程获取到了 cpu 的资源，开始执行方法。uuid=v2
set(lock,uuid);

4. index1 执行删除，此时会把 index2 的 lock 删除。index1 因为已经在方法中了，所以不需要重新上锁。index1 有执行的权限。index1 已经比较完成了，这个时候开始执行

```
redisTemplate.delete( key: "lock");
```

删除的是index2的锁！

解决：采用lua脚本保证操作的原子性

13.4.6 优化之LUA脚本保证删除的原子性

```
@GetMapping("testLockLua")
public void testLockLua() {
    //1 声明一个 uuid，将做为一个 value 放入我们的 key 所对应的值中
    String uuid = UUID.randomUUID().toString();
    //2 定义一个锁：lua 脚本可以使用同一把锁，来实现删除！
```

```

String skuId = "25"; // 访问 skuId 为 25 号的商品 100008348542
String lockKey = "lock:" + skuId; // 锁住的是每个商品的数据
// 3 获取锁
Boolean lock = redisTemplate.opsForValue().setIfAbsent(lockKey,
uuid, 3,
TimeUnit.SECONDS);
// 第一种： lock 与过期时间中间不写任何的代码。
// redisTemplate.expire("lock",10, TimeUnit.SECONDS); //设置过期时间
// 如果 true
if (lock) {
// 执行的业务逻辑开始
// 获取缓存中的 num 数据
Object value = redisTemplate.opsForValue().get("num");
// 如果是空直接返回
if (StringUtils.isEmpty(value)) {
return;
}
// 不是空 如果说在这出现了异常！ 那么 delete 就删除失败！也就是说锁永远存在！
int num = Integer.parseInt(value + "");
// 使 num 每次+1 放入缓存
redisTemplate.opsForValue().set("num", String.valueOf(++num));
/*使用 Lua 脚本来锁*/
// 定义 lua 脚本
String script = "if redis.call('get', KEYS[1]) == ARGV[1] then
return
redis.call('del', KEYS[1]) else return 0 end";
// 使用 redis 执行 lua 执行
DefaultRedisScript<Long> redisscript = new
DefaultRedisScript<>();
redisscript.setScriptText(script);
// 设置一下返回值类型 为 Long
// 因为删除判断的时候，返回的 0,给其封装为数据类型。如果不封装那么默认
为String类型
// 那么返回字符串与 0 会有发生错误。
redisscript.setResultType(Long.class);
// 第一个要是 script 脚本， 第二个需要判断的 key，第三个就是 key 所
对应的值。
redisTemplate.execute(redisscript, Arrays.asList(lockKey),
uuid);
} else {
// 其他线程等待
try {
// 睡眠
Thread.sleep(1000);
// 睡醒了之后，调用方法。
testLockLua();
} catch (InterruptedException e) {
e.printStackTrace();
}
}
}

```

```
        }
    }
}
```

LUA脚本详解：

客户端执行以上的命令：

- 如果服务器返回 `OK`，那么这个客户端获得锁。
- 如果服务器返回 `NIL`，那么客户端获取锁失败，可以在稍后再重试。

设置的过期时间到达之后，锁将自动释放。

可以通过以下修改，让这个锁实现更健壮：

- 不使用固定的字符串作为键的值，而是设置一个不可猜测（non-guessable）的长随机字符串，作为口令串（token）。
- 不使用 `DEL` 命令来释放锁，而是发送一个 Lua 脚本，这个脚本只在客户端传入的值和键的口令串相匹配时，才对键进行删除。

这两个改动可以防止持有过期锁的客户端误删现有锁的情况出现。

以下是一个简单的解锁脚本示例：

```
if redis.call("get",KEYS[1]) == ARGV[1]
then
    return redis.call("del",KEYS[1])
else
    return 0
end
```

项目中正确使用：

1. 定义 key，key 应该是为每个 sku 定义的，也就是每个 sku 有一把锁。

```
String lockKey ="lock:"+skuId; // 锁住的是每个商品的数据
Boolean lock = redisTemplate.opsForValue().setIfAbsent(lockKey,
uuid,3,TimeUnit.SECONDS);
```

2. /* 使用Lua脚本来锁 */
// 定义Lua 脚本
String script="if redis.call('get', KEYS[1]) == ARGV[1] then return redis.call('del', KEYS[1]) else return 0 end";
// 使用redis执行Lua脚本
// 第一种传值
// DefaultRedisScript<Object> redisScript = new DefaultRedisScript<>(script);
DefaultRedisScript<Long> redisScript = new DefaultRedisScript<>();
// 第二种传值
redisScript.setScriptText(script);
// 设置一下返回值类型 为Long
// 因为删除判断的时候，返回的0，将其转为数据类型。如果不封装默认返回String 类型，那么返回字符串与0 会有发生错误。
redisScript.setResultType(Long.class);
// 第一个要是script 脚本，第二个需要判断的key，第三个就是key所对应的值。
redisTemplate.execute(redisScript, Arrays.asList(lockKey),uuid);

13.4.7 总结

1. 加锁

```
String uuid = UUID.randomUUID().toString();
Boolean lock = this.redisTemplate.opsForValue()
.setIfAbsent("lock", uuid, 2, TimeUnit.SECONDS);
```

2. 使用lua释放锁

```
// 2. 释放锁 del
String script = "if redis.call('get', KEYS[1]) == ARGV[1] then
return
redis.call('del', KEYS[1]) else return 0 end";
// 设置 lua 脚本返回的数据类型
DefaultRedisScript<Long> redisscript = new
DefaultRedisScript<>();
// 设置 lua 脚本返回类型为 Long
redisscript.setResultType(Long.class);
redisscript.setScriptText(script);
redisTemplate.execute(redisscript,
Arrays.asList("lock"),uuid);
```

3. 重试获得锁

```
Thread.sleep(500);
testlock();
```

为了确保分布式锁可用，我们至少要**确保锁的实现同时满足以下四个条件**：

- **互斥性。**在任意时刻，只有一个客户端能持有锁。
- **不会发生死锁。**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。
- 解铃还须系铃人。**加锁和解锁必须是同一个客户端**，客户端自己不能把别人加的锁给解了。
- 加锁和解锁必须具有**原子性**。

十四、Redis6新功能

14.1 ACL

14.1.1 什么是ACL

Access Control List(访问控制列表)的缩写，该功能允许根据可以执行的命令和可以访问的键来限制某些连接

在 Redis 5 版本之前，Redis 安全规则只有密码控制 还有通过 rename 来调整高危命令比如 flushdb , KEYS* , shutdown 等。

Redis 6 则提供 ACL 的功能对用户进行更细粒度的权限控制：

1. 接入权限:用户名和密码

2. 可以执行的命令

3. 可以操作的 KEY

参考官网: <https://redis.io/topics/acl>

14.1.2 命令

1、使用 `acl list` 命令展现用户权限列表

```
127.0.0.1:6379> acl list
1) "user default on nopass ~* +@all"
```

用户名 是否启用 (on/off) 密码 (没密码 nopass) 可操作的key 可执行的命令

2、使用 `acl cat` 命令

(1) 查看添加权限指令类别

```
127.0.0.1:6379> acl cat
1) "keyspace"
2) "read"
3) "write"
4) "set"
5) "sortedset"
6) "list"
7) "hash"
8) "string"
9) "bitmap"
10) "hyperloglog"
11) "geo"
12) "stream"
13) "pubsub"
14) "admin"
15) "fast"
16) "slow"
17) "blocking"
18) "dangerous"
19) "connection"
20) "transaction"
21) "scripting"
```

(2) 加参数类型名可以查看类型下具体命令

```

127.0.0.1:6379> acl cat set
1) "scard"
2) "sort"
3) "smembers"
4) "sunionstore"
5) "sdiffstore"
6) "spop"
7) "sdiff"
8) "smismember"
9) "sscan"
10) "sismember"
11) "sunion"
12) "sinter"
13) "srem"
14) "srandmember"
15) "sinterstore"
16) "smove"
17) "sadd"

```

3、使用 `acl setuser` 命令创建和编辑用户ACL

(1) ACL 规则

下面是有效 ACL 规则的列表。某些规则只是用于激活或删除标志，或对用户 ACL 执行给定更改的单个单词。其他规则是字符前缀，它们与命令或类别名称、键模式等连接在一起。

ACL 规则		
类型	参数	说明
启动和禁用用户	on	激活某用户账号
	off	禁用某用户账号。注意，已验证的连接仍然可以工作。如果默认用户被标记为 off，则新连接将在未进行身份验证的情况下启动，并要求用户使用 AUTH 选项发送 AUTH 或 HELLO，以便以某种方式进行身份验证。
权限的添加删除	+<command>	将指令添加到用户可以调用的指令列表中
	-<command>	从用户可执行指令列表移除指令
	+@<category>	添加该类别中用户要调用的所有指令，有效类别为 @admin、@set、@sortedset… 等，通过调用 ACL CAT 命令查看完整列表。特殊类别 @all 表示所有命令，包括当前存在于服务器中的命令，以及将来将通过模块加载的命令。
	-@<category>	从用户可调用指令中移除类别
	allcommands	+@all 的别名
可操作键的添加或删除	nocommand	-@all 的别名
	~<pattern>	添加可作为用户可操作的键的模式。例如 ~* 允许所有的键

(2) 通过命令创建新用户默认权限

```
acl setuser 用户名
```

```
127.0.0.1:6379> acl setuser lebrwcd
OK
127.0.0.1:6379> acl list
1) "user default on nopass ~* &* +@all"
2) "user lebrwcd off &* -@all"
```

在上面的示例中，我根本没有指定任何规则。如果用户不存在，这将使用 just created 的默认属性来创建用户。如果用户已经存在，则上面的命令将不执行任何操作。

(3) 设置有用用户名、密码、ACL权限，并启用的用户

```
acl setuser 用户名 on >password ~cached:/* +get
```

表示该用户只能操作cached开头的键，只能进行get操作

```
127.0.0.1:6379> acl setuser user2 on >password ~cached:/* +get
OK
127.0.0.1:6379> acl list
1) "user default on nopass ~* +@all"
2) "user user1 off -@all"
3) "user user2 on #5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8
~cached:/* -@all +get"
```

(4) 切换用户，验证权限

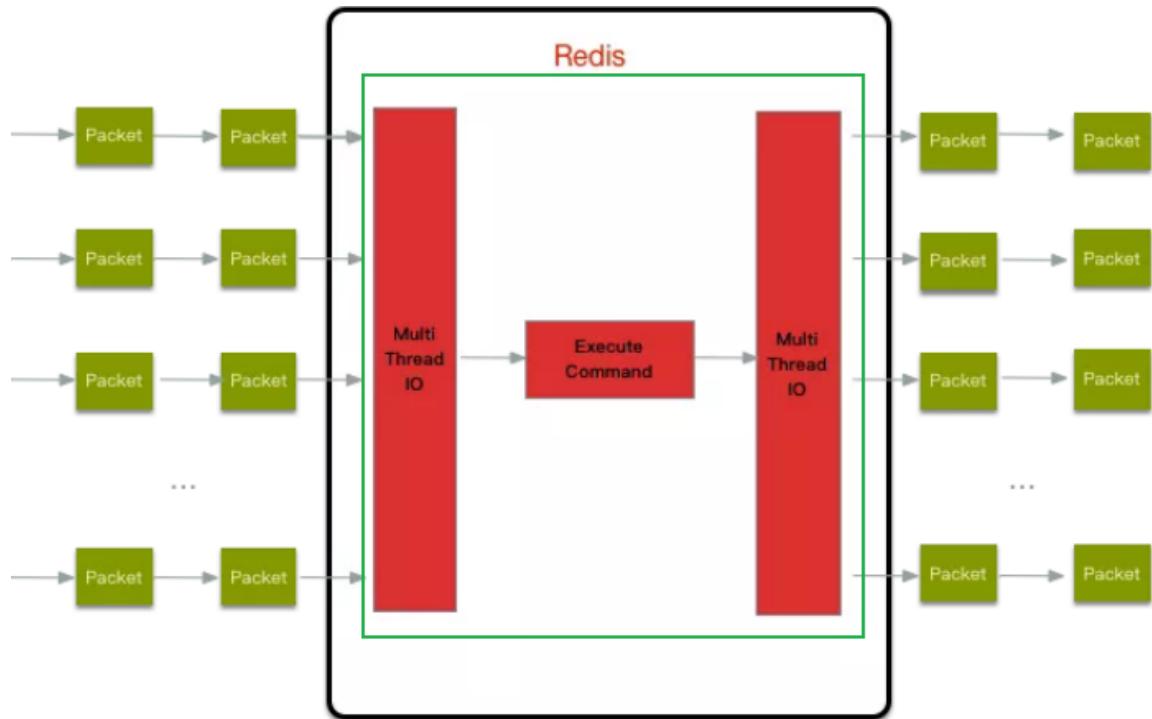
```
127.0.0.1:6379> acl whoami
"default"
127.0.0.1:6379> auth user2 password
OK
127.0.0.1:6379> acl whoami
(error) NOPERM this user has no permissions to run the 'acl' command or its subcommands
127.0.0.1:6379> get foo
(error) NOPERM this user has no permissions to access one of the keys used as arguments
127.0.0.1:6379> get cached:1121
(nil)
127.0.0.1:6379> set cached:1121 1121
(error) NOPERM this user has no permissions to run the 'set' command or its subcommands
```

14.2 IO多线程

Redis6 终于支撑多线程了，告别单线程了吗？IO 多线程其实指客户端交互部分的网络 IO 交互处理模块多线程，而非执行命令多线程。**Redis6 执行命令依然是单线程。**

14.2.1 原理架构

Redis 6 加入多线程，但跟 Memcached 这种从 IO 处理到数据访问多线程的实现模式有些差异。**Redis 的多线程部分只是用来处理网络数据的读写和协议解析，执行命令仍然是单线程。**之所以这么设计是不想因为多线程而变得复杂，需要去控制 key、lua、事务，LPUSH/LPOP 等等的并发问题。整体的设计大体如下：



另外，多线程 IO 默认也是不开启的，需要再配置文件中配置

```
io-threads-do-reads yes //开启多线程  
io-threads 4 //线程数
```

14.3 工具支持Cluster

之前老版 Redis 想要搭集群需要单独安装 ruby 环境，Redis 5 将 redis-trib.rb 的功能集成到 redis-cli。另外官方 redis-benchmark 工具开始支持 cluster 模式了，通过多线程的方式对多个分片进行压测。

14.4 Redis新功能持续关注

Redis6 新功能还有：

- 1、**RESP3 新的 Redis 通信协议**：优化服务端与客户端之间通信
- 2、**Client side caching 客户端缓存**：基于 RESP3 协议实现的客户端缓存功能。为了进一步提升缓存的性能，将客户端经常访问的数据 cache 到客户端。减少 TCP 网络交互。
- 3、**Proxy 集群代理模式**：Proxy 功能，让 Cluster 拥有像单实例一样的接入方式，降低大家使用 cluster 的门槛。不过需要注意的是代理不改变 Cluster 的功能限制，不支持的命令还是不会支持，比如跨 slot 的多 Key 操作。

4、**Modules API**

Redis 6 中模块 API 开发进展非常大，因为 Redis Labs 为了开发复杂的功能，从一开始就用上 Redis 模块。Redis 可以变成一个框架，利用 Modules 来构建不同系统，而不需要从头开始写然后还要 BSD 许可。Redis 一开始就是一个向编写各种系统开放的平台