# 01【熟悉】NoSQL的简介

## 1，什么是NoSQL

　    NoSQL 是 Not Only SQL 的缩写，意即"不仅仅是SQL"的意思，泛指非关系型的数据库。强调Key-Value Stores和文档数据库的优点。

　　NoSQL产品是传统关系型数据库的功能阉割版本，通过减少用不到或很少用的功能，来大幅度提高产品性能

## 2，NoSQL起源

过去，关系型数据库(SQL Server、Oracle、MySQL)是数据持久化的唯一选择，但随着发展，关系型数据库存在以下问题。

#### 问题1：不能满足高性能查询需求

  我们使用:Java、.Net等语言编写程序，是面向对象的。但用数据库都是关系型数据库。存储结构是面向对象的，但是数据库却是关系的，所以在每次存储或者查询数据时，我们都需要做转换。类似Hibernate、Mybatis这样的ORM框架确实可以简化这个过程，但是在对高性能查询需求时，这些ORM框架就捉襟见肘了。

#### 问题2：应用程序规模的变大

  网络应用程序的规模变大，需要储存更多的数据、服务更多的用户以及需求更多的计算能力。为了应对这种情形，我们需要不停的扩展。

  扩展分为两类：一种是纵向扩展，即购买更好的机器，更多的磁盘、更多的内存等等。另一种是横向扩展，即购买更多的机器组成集群。在巨大的规模下，纵向扩展发挥的作用并不是很大。首先单机器性能提升需要巨额的开销并且有着性能的上限，在Google和Facebook这种规模下，永远不可能使用一台机器支撑所有的负载。鉴于这种情况，我们需要新的数据库，因为关系数据库并不能很好的运行在集群上

## 3，NoSQL数据库类型

#### ①键值（Key-Value）数据库[Redis/Memcached]

适用场景：

储存用户信息，比如会话、配置文件、参数、购物车等等。这些信息一般都和ID（键）挂钩，这种情景下键值数据库是个很好的选择。

不适用场景：

1.取代通过键查询，而是通过值来查询。Key-Value数据库中根本没有通过值查询的途径

2.需要储存数据之间的关系。在Key-Value数据库中不能通过两个或以上的键来关联数据。

3.事务的支持。在Key-Value数据库中故障产生时不可以进行回滚。

#### ②面向文档（Document-Oriented）数据库[MongoDB]

数据可以使用XML、JSON或者JSONB等多种形式存储。

适用场景：1.日志 2.分析

不适用场景：不支持事务

#### ③列存储（Wide Column Store/Column-Family）数据库[HBASE]

列存储数据库将数据储存在列族（column family）中，一个列族存储经常被一起查询的相关数据。举个例子，如果我们有一个Person类，我们通常会一起查询他们的姓名和年龄而不是薪资。这种情况下，姓名和年龄就会被放入一个列族中，而薪资则在另一个列族中。

适用场景：1.日志 2.博客平台,我们储存每个信息到不同的列族中。举个例子，标签可以储存在一个，类别可以在一个，而文章则在另一个。

不适用场景：1.ACID事务 2.原型设计。在模型设计之初，我们根本不可能去预测它的查询方式，而一旦查询方式改变，我们就必须重新设计列族。

#### ④图（Graph-Oriented）数据库[Neo4J]

适用范围很小，主要用用网络拓扑分析 如脉脉的人员关系图等

## 4，传统RDBMS VS NOSQL

 RDBMS

- 高度组织化结构化数据

- 结构化查询语言（SQL）

- 数据和关系都存储在单独的表中。

- 数据操纵语言，数据定义语言

- 严格的一致性

- 基础事务

 NoSQL

- 代表着不仅仅是SQL

- 没有声明性查询语言

- 没有预定义的模式

- 键 - 值对存储，列存储，文档存储，图形数据库

- 最终一致性，而非ACID【原子，一致，隔离，持久】属性

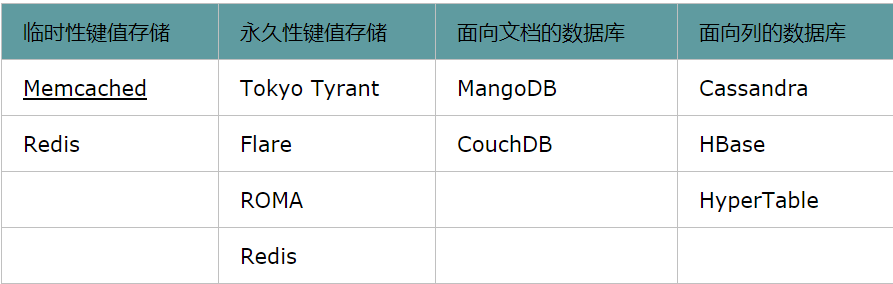
- 非结构化和不可预知的数据

- CAP定理【一致性，可用性，容错性】

- 高性能，高可用性和可伸缩性

# 02【熟悉】常见的NoSQL及区别

## 1，常见的NoSQL数据库



## 2，区别

**1，Memcached**

挥发性(临时性)的键值存储

一般作为关系型数据库的缓存来使用

具有非常快的处理速度

由于存在数据丢失的可能，所以一般用来处理不需要持久保存的数据

用于需要使用expires时(需要定期清除数据)

使用一致性散列(Consistent Hashing)算法来分散数据

**2，Tokyo Tyrant**

持久性的键值存储

用来处理需要持久保存，高速处理的数据

具有非常快的处理速度

用于不需要定期清除的数据

使用一致性散列(Consistent Hashing)算法来分散数据

**3，Redis**

兼具Memcached和Tokyo Tyrant优势的键值存储

擅长处理数组类型的数据

具有非常快的处理速度

可以高速处理时间序列的数据，易于处理集合运算

拥有很多可以进行原子操作的方法

使用一致性散列(Consistent Hashing)算法来分散数据

**4，MongoDB**

面向无需定义表结构的文档数据

具有非常快的处理速度

通过BSON的形式可以保存和查询任何类型的数据

无法进行JOIN处理，但是可以通过嵌入(embed)来实现同样的功能

使用sharding(范围分割)算法来分散数据

# 03【熟悉】Redis简介

## 1，Redis简介

         Redis:Remote Dictionary Server(远程字典服务器)

         Redis是当前比较热门的NOSQL系统之一，它是一个开源的使用ANSI c语言编写的key-value存储系统（区别于MySQL的二维表格的形式存储。）。和Memcache类似，但很大程度补偿了Memcache的不足。和Memcache一样，Redis数据都是缓存在计算机内存中，不同的是，Memcache只能将数据缓存到内存中，无法自动定期写入硬盘，这就表示，一断电或重启，内存清空，数据丢失。所以Memcache的应用场景适用于缓存无需持久化的数据。而Redis不同的是它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，实现数据的持久化

## 2，Redis的特点

1. Redis读取的速度是110000次/s，写的速度是81000次/s
2. 原子 。Redis的所有操作都是原子性的，同时Redis还支持对几个操作全并后的原子性执行。
3. 支持多种数据结构：string（字符串）；list（列表）；hash（哈希），set（集合）；zset(有序集合)
4. 持久化，主从复制（集群）
5. 支持过期时间，支持事务，消息订阅。
6. 官方不支持window,但是又第三方版本。

**3，Redis的应用场景**

**1、数据缓存（提高访问性能）**

将一些数据在短时间之内不会发生变化，而且它们还要被频繁访问，为了提高用户的请求速度和降低网站的负载，降低数据库的读写次数，就把这些数据放到缓存中。

**2、会话缓存**

（session cache，保存web会话信息）

**3、排行榜/计数器**

（NGINX+lua+redis计数器进行IP自动封禁）

**4、消息队列**

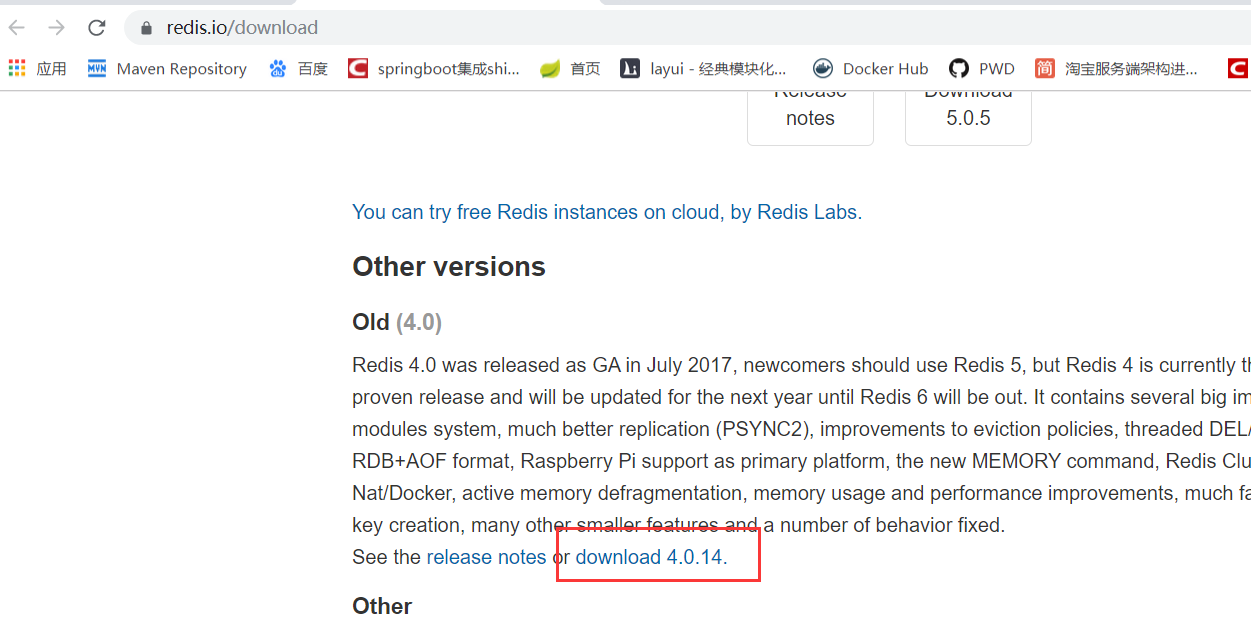
（构建实时消息系统，聊天，群聊）

# 04【掌握】Redis的安装及启动停止

## 1，下载Redis4

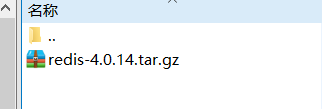
redis目前的最新版本是5.X此教程是以4.X为基础来整理的，大家尽量保持一致

<https://redis.io/download>





## 2，打开VM虚拟机把文件copy到software里面



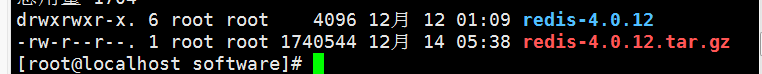
## 3，开始安装

### 1，安装gcc  目地是编译软件

yum install gcc-c++

### 2，解压

tar -zxvf redis-4.0.12.tar.gz



### 2，把解压的文件copy到/usr/local/src里面

cp -r /root/software/redis-4.0.12 /usr/local/src/

改名

cd /usr/local/src/

mv redis-4.0.12 redis

### 3，打开/usr/local/src/redis/deps进行编译依赖项

cd /usr/local/src/redis/deps

make hiredis lua jemalloc linenoise

### 4，打开/usr/local/src/redis进行编译

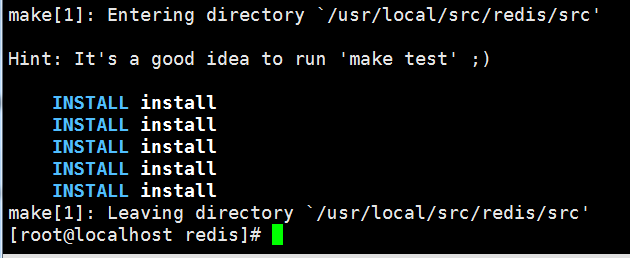
cd /usr/local/src/redis

make

### 5，在上面的Redis目录安装把它安装到/usr/local/redis里面

mkdir /usr/local/redis

make install PREFIX=/usr/local/redis



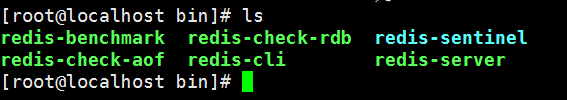
看到上面的说明安装成功了哦

### 6，验证安装是否成功

cd /usr/local/redis/bin

ls

看到如下启动文件就可以了



使用which命令查看系统里面是否有redis的服务

which redis-server



显示为没有

### 7，把配置文件移动到/root/myredis目录[目录可以自定义] 可以为/root/myredis

使用which命令查看系统里面是否有redis的服务

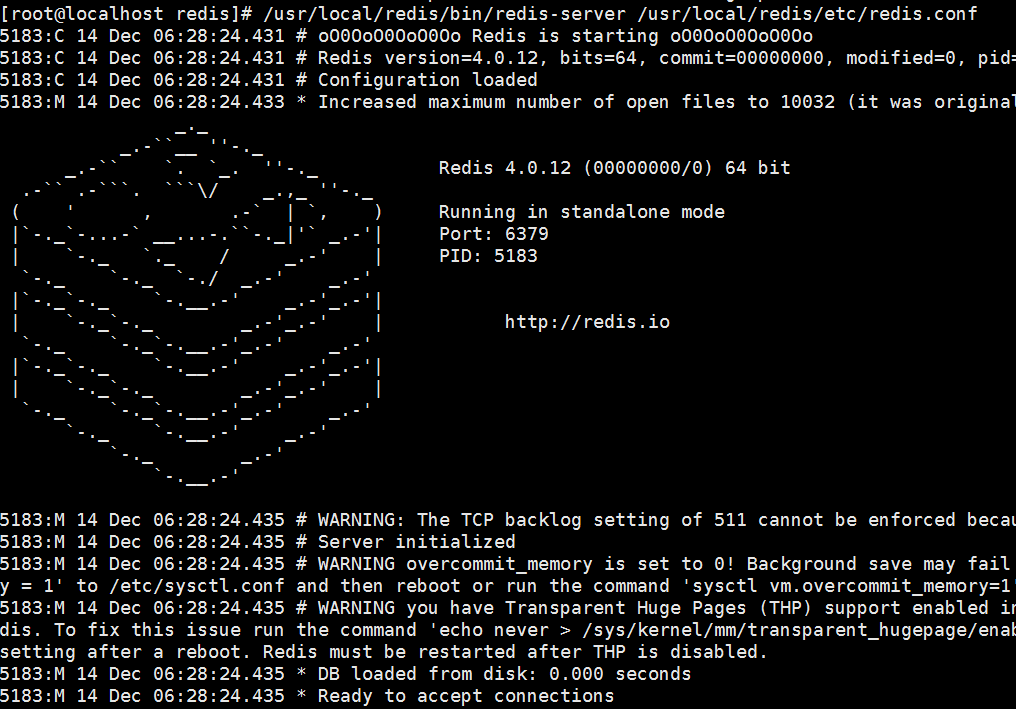
mkdir /root/myredis

cp /usr/local/src/redis/redis.conf /root/myredis

### 8，启动Redis

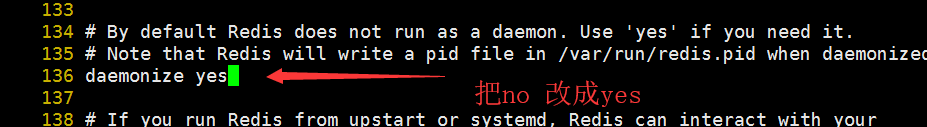
cd /usr/local/redis/bin

./redis-server /root/myredis/redis.conf



### 9，默认情况，Redis不是在后台运行，我们需要把redis放在后台运行

vi /root/myredis/redis.conf

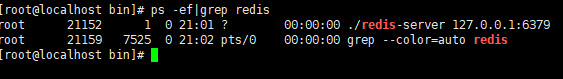


### 10，再次启动查看进程

./redis-server /usr/myredis/redis.conf

#查看进程

ps -ef|grep redis



可以看到在6379端口号已启动了redis

### 11，客户端链接和退出

#连接

cd /usr/local/redis/bin

./redis-cli 默认是-h 127.0.0.1 -p 6379

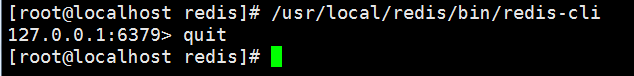
#退出

quit

​

ping

PONG



### 12，停止redis

cd /usr/local/redis/bin

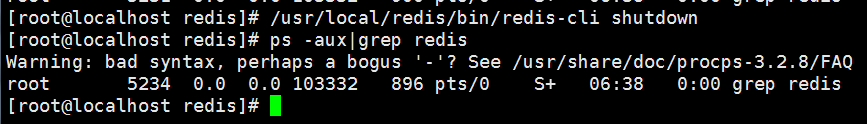
./redis-cli shutdown

#或者

pkill redis-server

#再次查看进程

ps -ef|grep redis



### 13，开机自启Redis的配置

　　vim /etc/rc.local

　　加入

　　/usr/local/redis/bin/redis-server /root/myredis/redis-conf

### 14，bin目录的文件说明

　　redis-benchmark：redis性能测试工具

　　redis-check-aof：检查aof日志的工具

　　redis-check-dump：检查rdb日志的工具

　　redis-cli：连接用的客户端

　　redis-server：redis服务进程

## 3，redis配置[后面细说]

daemonize：如需要在后台运行，把该项的值改为yes

pdifile：把pid文件放在/var/run/redis.pid，可以配置到其他地址

bind：指定redis只接收来自该IP的请求，如果不设置，那么将处理所有请求，在生产环节中最好设置该项

port：监听端口，默认为6379

timeout：设置客户端连接时的超时时间，单位为秒

loglevel：等级分为4级，debug，revbose，notice和warning。生产环境下一般开启notice

logfile：配置log文件地址，默认使用标准输出，即打印在命令行终端的端口上

database：设置数据库的个数，默认使用的数据库是0

save：设置redis进行数据库镜像的频率

rdbcompression：在进行镜像备份时，是否进行压缩

dbfilename：镜像备份文件的文件名

dir：数据库镜像备份的文件放置的路径

slaveof：设置该数据库为其他数据库的从数据库

masterauth：当主数据库连接需要密码验证时，在这里设定

requirepass：设置客户端连接后进行任何其他指定前需要使用的密码

maxclients：限制同时连接的客户端数量

maxmemory：设置redis能够使用的最大内存

appendonly：开启appendonly模式后，redis会把每一次所接收到的写操作都追加到appendonly.aof文件中，当redis重新启动时，会从该文件恢复出之前的状态

appendfsync：设置appendonly.aof文件进行同步的频率

vm\_enabled：是否开启虚拟内存支持

vm\_swap\_file：设置虚拟内存的交换文件的路径

vm\_max\_momery：设置开启虚拟内存后，redis将使用的最大物理内存的大小，默认为0

vm\_page\_size：设置虚拟内存页的大小

vm\_pages：设置交换文件的总的page数量

vm\_max\_thrrads：设置vm IO同时使用的线程数量

# 05【掌握】启动后相关知识串讲

## 1，单进程单线程

采用多路 I/O 复用技术可以让单个线程高效的处理多个连接请求（尽量减少网络IO的时间消耗）

why?

多线程处理可能涉及到锁

多线程处理会涉及到线程切换而消耗CPU

单进程不存在线程安全问题

缺点：

无法发挥多核CPU性能，不过可以通过在单机开多个Redis实例来完善

## 2，默认16个兄弟一起站台

默认16个数据库，类似数组下表从零开始，初始默认使用零号库

## 3，切换数据库命令

select命令切换数据库

## 4，常用基本命令

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

## 5，其它说明

统一密码管理，16个库都是同样密码，要么都OK要么一个也连接不上

Redis索引都是从零开始

## 6,为什么默认端口是6379

先介绍下redis的作者Salvatore Sanfilippo(Antirez)，意大利人，就是下图这位。



Antirez现在已经40多岁了，依然奋斗在代码一线，为开源社区做贡献。Antirez出生在非英语系国家，所以英语一直是他的短板。他曾经写过一篇博文，《英语伤痛 15 年》，以自己的实际经历鼓励非英语系国家的程序员突破英语障碍。或说回来，在他的另一篇博文《Redis as an LRU cache 》中，写到了为什么选用6379端口：

用一张图片来翻译一下，6379 就是这个意思：



而Merz全名Alessia Merz，是意大利的一位广告女郎，就是下面这位：



在Antirez看来，这个名字是愚蠢的代名词，所以就选了这个6379

这个还真不知道有没有其它原因（自行脑补）

# 06【掌握】redis的数据类型

## 1，概述

        使用Redis进行应用设计和开发的一个核心概念是数据类型。与关系数据库不同，在Redis中不存在需要我们担心的表，在使用Redis进行应用设计和开发时，我们首先应该考虑的是Redis原生支持的哪种数据类型阳适合我们的应该场景，此外，我们无法像在关系数据库中那样，使用sql来操作Redis中的数据，相反，我们需要直接使用API发送数据反对应的命令，来操作想要操作的数据

## 2，字符串类型

    字符串类型是编程语言和应用程序中最常见和最有用的数据类型，也是Redis的基本数据类型之一，事实上，Redis中所有键都必须是字符串。

## 3，list数据类型

    列表是应用我只是应该程序开发中非常有用的数据类型之一，列表能存在一组对象，因此它也可以被用于栈或者队列，在Redis中，与键相关的联的值可以是字符串组成的列表，Redis中的列表更像是数据结构中的双向链表。

## 4，hash数据类型

    哈希表示字段和值之间的映射关系，与JAVA中的Map类似，Redis数据集本身就可以看做一个哈希，其中字符串类型的键关联到如字符串和列表之类的数据对象，而Reidis的数据对象也可以再次使用哈希，其字段和值必须 是字符串。

## 5，set数据类型

    集合类型是由唯一，无序对象组成的集合(collection).它经常用于测试某个成员是集合中，重复项删除和集合运算（求并，交，差集），Redis的值对象可以是字符串集合。

## 6，zset(sortset)数据类型

    有序集合是一个类似于set但是更复杂的数据类型,单词sorted意为着这种集合中的每个元素都有一个可用于排序的权重，并且我们可以按顺序从集合中得到元素在某些需要一个保持数据有序的场景中，使用这种原生的序的特性是很方便的。

# 07【掌握】Redis的相关命令详解

关于命令的学习查询看这个网站http://www.redis.net.cn/order/

或者：http://redisdoc.com/

## 1，常用命令

keys \* 获取所有的key

select 0 选择第一个库

move myString 1 将当前的数据库key移动到某个数据库,目标库有，则不能移动

flushdb 清除指定库

randomkey 从当前数据库中随机返回

type key 类型

del key1 删除key

exists key 判断是否存在key

expire key 10 10过期

pexpire key 1000 毫秒

persist key 删除过期时间

ttl key 查看还有多少秒过期，-1表示永不过期，-2表示已过期

## 2，string类型相关命令

set name cxx

get name

getrange name 0 -1 字符串分段 0 -1是全部 0 -2 ==n-1

getset name new\_cxx 设置值，返回旧值

mset key1 key2 批量设置

mget key1 key2 批量获取

setnx key value 不存在就插入（not exists）

setrange key index value 从index开始替换value

incr age 递增

incrby age 10 递增

decr age 递减

decrby age 10 递减

incrbyfloat 增减浮点数

append 追加

strlen 长度

object encoding key 得到key 的类型 string里面有三种编码

int 用于能够副作用64位有符号整数表示的字符串

embstr 用于长度小于或等于44字节 Redis3.x中是39字节，这种类型的编码在内存使用时性能更好

raw 用于长度大于44字节的

## 3，list

lpush mylist a b c 左插入

rpush mylist x y z 右插入

lrange mylist 0 -1 取出数据集合 0 -1是取出所有 0 1取第第一个和第二个

lpop mylist 弹出集合最后一个元素 弹出之后就没有了哦

rpop mylist 弹出第一个元素 弹出之后就没有了哦

llen mylist 长度

lrem mylist count value 删除

|-COUNT 的值可以是以下几种：

|--count > 0 : 从表头开始向表尾搜索，移除与 VALUE 相等的元素，数量为 COUNT 。

|--count < 0 : 从表尾开始向表头搜索，移除与 VALUE 相等的元素，数量为 COUNT 的绝对值。

|--count = 0 : 移除表中所有与 VALUE 相等的值。

lindex mylist 2 指定索引的值

lset mylist 2 n 索引设值

ltrim mylist 0 4

|--对一个列表进行修剪(trim)，就是说，让列表只保留指定区间内的元素，不在指定区间之内的元素都将被删除。

下标 0 表示列表的第一个元素，以 1 表示列表的第二个元素，以此类推。

你也可以使用负数下标，以 -1 表示列表的最后一个元素， -2 表示列表的倒数第二个元素，以此类推。

linsert mylist before a 插入

linsert mylist after a 插入

|--命令用于在列表的元素前或者后插入元素。 当指定元素不存在于列表中时，不执行任何操作。

当列表不存在时，被视为空列表，不执行任何操作。 如果 key 不是列表类型，返回一个错误。

rpoplpush list list2 转移列表的数据

|--命令用于移除列表的最后一个元素，并将该元素添加到另一个列表并返回。

## 4，hash

hset myhash name cxx

|--命令用于为哈希表中的字段赋值 。

|--如果哈希表不存在，一个新的哈希表被创建并进行 HSET 操作。

|--如果字段已经存在于哈希表中，旧值将被覆盖。

hget myhash name

hmset myhash name cxx age 25 note "i am notes"

hmget myhash name age note

hgetall myhash 获取所有的

hexists myhash name 是否存在

hsetnx myhash score 100 设置不存在的 如果存在，不做处理

hincrby myhash id 1 递增

hdel myhash name 删除

hkeys myhash 只取key

hvals myhash 只取value

hlen myhash 长度

## 5，set

sadd myset redis

smembers myset 数据集合

srem myset set1 删除

sismember myset set1 判断元素是否在集合中

scard key\_name 个数

sdiff | sinter | sunion 操作：集合间运算：差集 | 交集 | 并集

srandmember 随机获取集合中的元素

spop 从集合中弹出一个元素

## 6，zset

zadd zset 1 one

zadd zset 2 two

zadd zset 3 three

zincrby zset 1 one 增长分数

zscore zset two 获取分数

zrange zset 0 -1 withscores 范围值

zrangebyscore zset 10 25 withscores 指定范围的值

zrangebyscore zset 10 25 withscores limit 1 2 分页

Zrevrangebyscore zset 10 25 withscores 指定范围的值

zcard zset 元素数量

Zcount zset 获得指定分数范围内的元素个数

Zrem zset one two 删除一个或多个元素

Zremrangebyrank zset 0 1 按照排名范围删除元素

Zremrangebyscore zset 0 1 按照分数范围删除元素

Zrank zset 0 -1 分数最小的元素排名为0

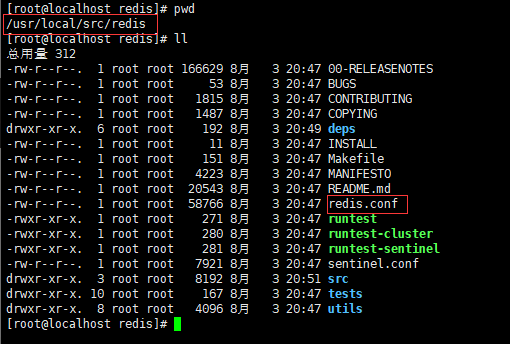
Zrevrank zset 0 -1 分数最大的元素排名为0

Zinterstore

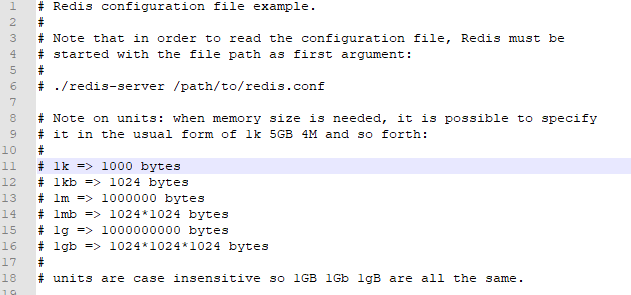
zunionstore rank:last\_week 7 rank:20150323 rank:20150324 rank:20150325 weights 1 1 1 1 1 1 1

# 08【熟悉】解析配置文件redis.conf

## 1，配置文件在哪



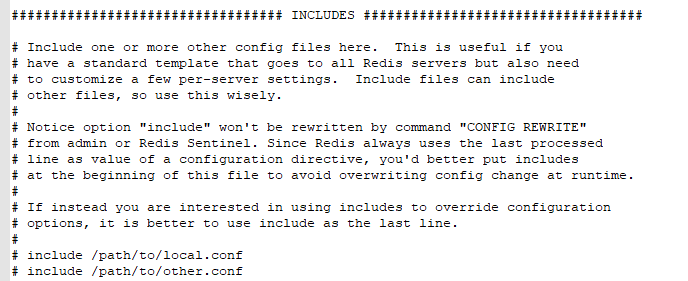
## 2，Units单位



1 配置大小单位,开头定义了一些基本的度量单位，只支持bytes，不支持bit

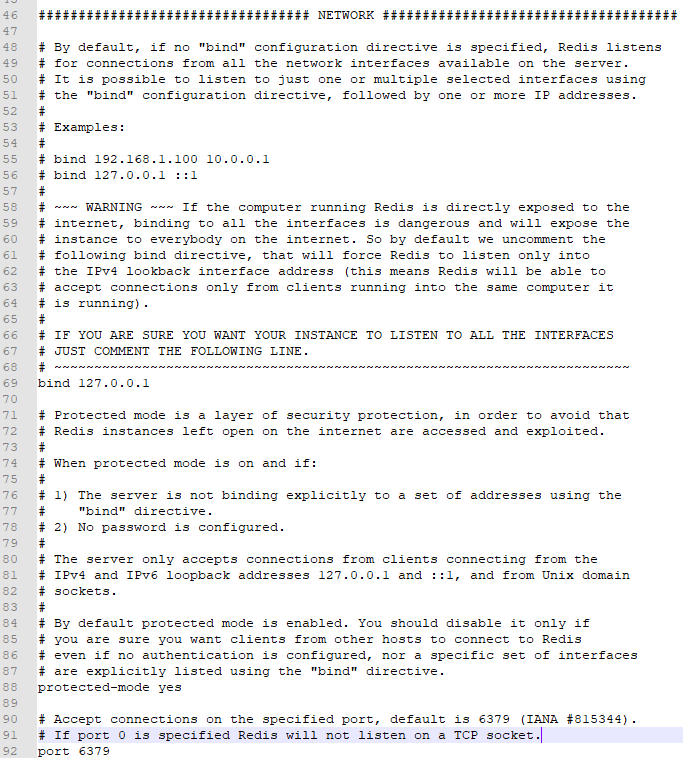
2 对大小写不敏感

## 3，INCLUDES包含



   和我们的spring配置文件类似，可以通过includes包含，redis.conf可以作为总闸，包含其他

## 4，NETWORK通用



默认情况下，如果没有指定“bind”配置指令，则Redis侦听用于连接服务器上所有可用的网络接口。

可以只监听一个或多个选择的接口 "bind"配置指令，后面跟着一个或多个IP地址。

1，bind

默认情况下，redis 在 server 上所有有效的网络接口上监听客户端连接。如果只想让它在一个或多个网络接口上监听，那你就绑定一个IP或者多个IP。多个ip空格分隔即可。



2，prot运行端口



3，Tcp-backlog

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，所以需要确认增大somaxconn和tcp\_max\_syn\_backlog两个值

来达到想要的效果

4，timeout



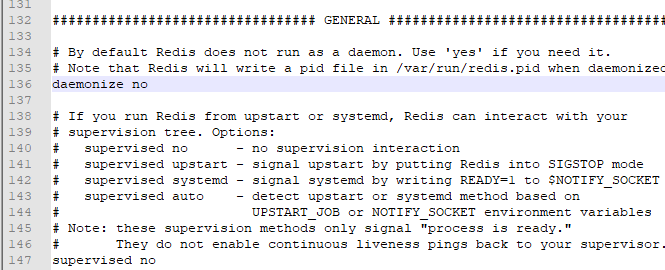
当客户端闲置多少秒后关闭连接，如果设置为0表示关闭该功能。

5，tcp-keepalive 300

单位是秒，表示将周期性的使用SO\_KEEPALIVE检测客户端是否还处于健康状态，避免服务器一直阻塞，官方给出的建议值是300S

  建议设置成60

## 5，GRNERAL通用



**daemonize no**是否以守护模式启动，默认为no，配置为yes时以守护模式启动，这时redis instance会将进程号pid写入默认文件/var/run/redis.pid。

**supervised no** 可以通过upstart和systemd管理Redis守护进程，这个参数是和具体的操作系统相关的。

**pidfile /var/run/redis\_6379.pid** 配置pid文件路径。当redis以守护模式启动时，如果没有配置pidfile，pidfile默认值是/var/run/redis.pid 。

**loglevel notice** 日志级别。可选项有：debug（记录大量日志信息，适用于开发、测试阶段）； verbose（较多日志信息）； notice（适量日志信息，使用于生产环境）；warning（仅有部分重要、关键信息才会被记录）。

**logfile ""**日志文件的位置，当指定为空字符串时，为标准输出，如果redis已守护进程模式运行，那么日志将会输出到 /dev/null 。

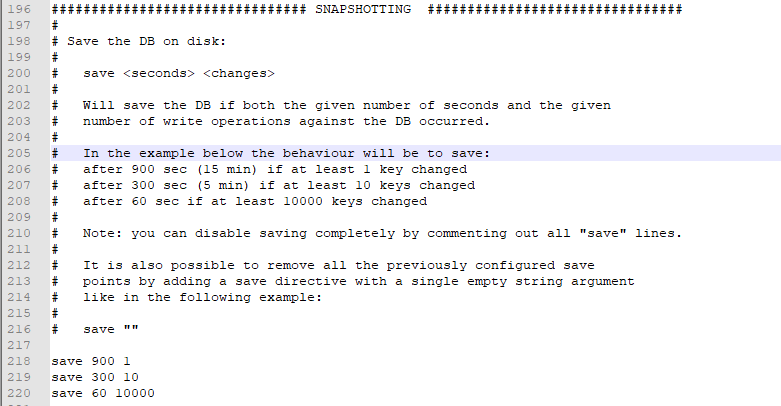
**syslog-enabled no**  是否把日志记录到系统日志。

**syslog-ident**  设置系统日志的id     如  syslog-ident redis

**databases 16**  设置数据库的数目。默认的数据库是DB 0 ，可以在每个连接上使用select <dbid> 命令选择一个不同的数据库，dbid是一个介于0到databases - 1 之间的数值。

**always-show-logo yes** 是否一直显示日志

## 6，SNAPSHOTTING快照



**save**  保存数据到磁盘。格式是：save <seconds> <changes> ，含义是在 seconds 秒之后至少有 changes个keys 发生改变则保存一次。

如        save 900 1      900秒有一条数据改变就保存

            save 300 10      300秒有10条数据改变就保存

            save 60 10000   600秒有10000条数据改变就保存

**stop-writes-on-bgsave-error**yes  默认情况下，如果 redis 最后一次的后台保存失败，redis 将停止接受写操作，这样以一种强硬的方式让用户知道数据不能正确的持久化到磁盘， 否则就会没人注意到灾难的发生。 如果后台保存进程重新启动工作了，redis 也将自动的允许写操作。然而你要是安装了靠谱的监控，你可能不希望 redis 这样做，那你就改成 no 好了。

**rdbcompression**yes  是否在dump .rdb数据库的时候压缩字符串，默认设置为yes。如果你想节约一些cpu资源的话，可以把它设置为no，这样的话数据集就可能会比较大。

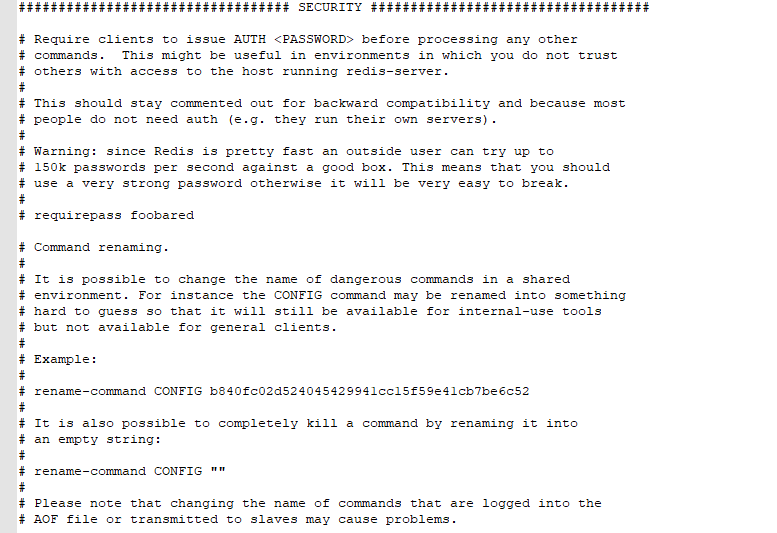
**rdbchecksum**yes  是否CRC64校验rdb文件，会有一定的性能损失（大概10%）

**dbfilename** dump.rdb   rdb文件的名字。

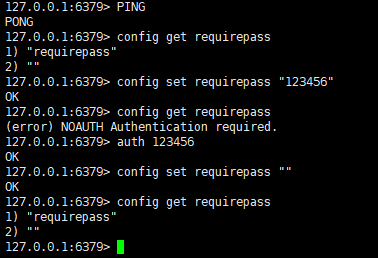
## 7，REPLICATION复制

讲主从复制再说

## 8，SECURITY安全



主要用于访问密码和查看，设置和取消



**9，LIMITS限制**

**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。当你

无法设置进程文件句柄限制时，redis会设置为当前的文件句柄限制值减去32，因为redis会为自

身内部处理逻辑留一些句柄出来。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。

**maxmemory**

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

**maxmemory-policy**

（1）volatile-lru：使用LRU算法移除key，只对设置了过期时间的键

（2）allkeys-lru：使用LRU算法移除key

（3）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

（4）allkeys-random：移除随机的key

（5）volatile-ttl：移除那些TTL值最小的key，即那些最近要过期的key

（6）noeviction：不进行移除。针对写操作，只是返回错误信息

**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

## 10，APPEND ONLY MODE 追加[持久化再说]

# 09【宝典】常见配置redis.conf介绍

参数说明

redis.conf 配置项说明如下：

1. Redis默认不是以守护进程的方式运行，可以通过该配置项修改，使用yes启用守护进程

  daemonize no

2. 当Redis以守护进程方式运行时，Redis默认会把pid写入/var/run/redis.pid文件，可以通过pidfile指定

  pidfile /var/run/redis.pid

3. 指定Redis监听端口，默认端口为6379，作者在自己的一篇博文中解释了为什么选用6379作为默认端口，因为6379在手机按键上MERZ对应的号码，而MERZ取自意大利歌女Alessia Merz的名字

  port 6379

4. 绑定的主机地址

  bind 127.0.0.1

5.当 客户端闲置多长时间后关闭连接，如果指定为0，表示关闭该功能

  timeout 300

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

  loglevel verbose

7. 日志记录方式，默认为标准输出，如果配置Redis为守护进程方式运行，而这里又配置为日志记录方式为标准输出，则日志将会发送给/dev/null

  logfile stdout

8. 设置数据库的数量，默认数据库为0，可以使用SELECT <dbid>命令在连接上指定数据库id

  databases 16

9. 指定在多长时间内，有多少次更新操作，就将数据同步到数据文件，可以多个条件配合

  save <seconds> <changes>

  Redis默认配置文件中提供了三个条件：

  save 900 1

  save 300 10

  save 60 10000

  分别表示900秒（15分钟）内有1个更改，300秒（5分钟）内有10个更改以及60秒内有10000个更改。

10. 指定存储至本地数据库时是否压缩数据，默认为yes，Redis采用LZF压缩，如果为了节省CPU时间，可以关闭该选项，但会导致数据库文件变的巨大

  rdbcompression yes

11. 指定本地数据库文件名，默认值为dump.rdb

  dbfilename dump.rdb

12. 指定本地数据库存放目录

  dir ./

13. 设置当本机为slav服务时，设置master服务的IP地址及端口，在Redis启动时，它会自动从master进行数据同步

  slaveof <masterip> <masterport>

14. 当master服务设置了密码保护时，slav服务连接master的密码

  masterauth <master-password>

15. 设置Redis连接密码，如果配置了连接密码，客户端在连接Redis时需要通过AUTH <password>命令提供密码，默认关闭

  requirepass foobared

16. 设置同一时间最大客户端连接数，默认无限制，Redis可以同时打开的客户端连接数为Redis进程可以打开的最大文件描述符数，如果设置 maxclients 0，表示不作限制。当客户端连接数到达限制时，Redis会关闭新的连接并向客户端返回max number of clients reached错误信息

  maxclients 128

17. 指定Redis最大内存限制，Redis在启动时会把数据加载到内存中，达到最大内存后，Redis会先尝试清除已到期或即将到期的Key，当此方法处理 后，仍然到达最大内存设置，将无法再进行写入操作，但仍然可以进行读取操作。Redis新的vm机制，会把Key存放内存，Value会存放在swap区

  maxmemory <bytes>

18. 指定是否在每次更新操作后进行日志记录，Redis在默认情况下是异步的把数据写入磁盘，如果不开启，可能会在断电时导致一段时间内的数据丢失。因为 redis本身同步数据文件是按上面save条件来同步的，所以有的数据会在一段时间内只存在于内存中。默认为no

  appendonly no

19. 指定更新日志文件名，默认为appendonly.aof

   appendfilename appendonly.aof

20. 指定更新日志条件，共有3个可选值：

  no：表示等操作系统进行数据缓存同步到磁盘（快）

  always：表示每次更新操作后手动调用fsync()将数据写到磁盘（慢，安全）

  everysec：表示每秒同步一次（折衷，默认值）

  appendfsync everysec

21. 指定是否启用虚拟内存机制，默认值为no，简单的介绍一下，VM机制将数据分页存放，由Redis将访问量较少的页即冷数据swap到磁盘上，访问多的页面由磁盘自动换出到内存中（在后面的文章我会仔细分析Redis的VM机制）

   vm-enabled no

22. 虚拟内存文件路径，默认值为/tmp/redis.swap，不可多个Redis实例共享

   vm-swap-file /tmp/redis.swap

23. 将所有大于vm-max-memory的数据存入虚拟内存,无论vm-max-memory设置多小,所有索引数据都是内存存储的(Redis的索引数据 就是keys),也就是说,当vm-max-memory设置为0的时候,其实是所有value都存在于磁盘。默认值为0

   vm-max-memory 0

24. Redis swap文件分成了很多的page，一个对象可以保存在多个page上面，但一个page上不能被多个对象共享，vm-page-size是要根据存储的 数据大小来设定的，作者建议如果存储很多小对象，page大小最好设置为32或者64bytes；如果存储很大大对象，则可以使用更大的page，如果不 确定，就使用默认值

   vm-page-size 32

25. 设置swap文件中的page数量，由于页表（一种表示页面空闲或使用的bitmap）是在放在内存中的，，在磁盘上每8个pages将消耗1byte的内存。

   vm-pages 134217728

26. 设置访问swap文件的线程数,最好不要超过机器的核数,如果设置为0,那么所有对swap文件的操作都是串行的，可能会造成比较长时间的延迟。默认值为4

   vm-max-threads 4

27. 设置在向客户端应答时，是否把较小的包合并为一个包发送，默认为开启

  glueoutputbuf yes

28. 指定在超过一定的数量或者最大的元素超过某一临界值时，采用一种特殊的哈希算法

  hash-max-zipmap-entries 64

  hash-max-zipmap-value 512

29. 指定是否激活重置哈希，默认为开启（后面在介绍Redis的哈希算法时具体介绍）

  activerehashing yes

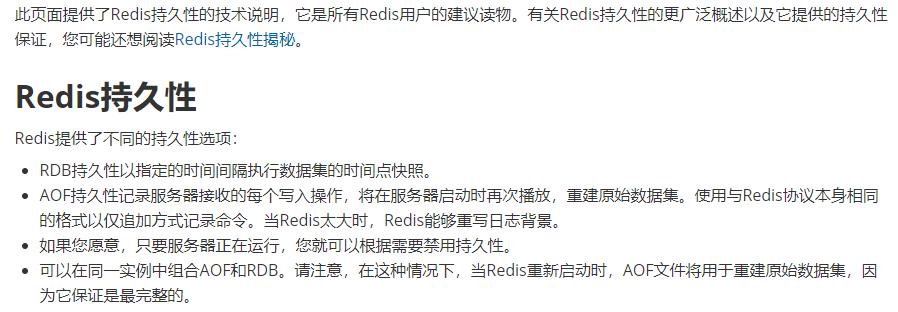
30. 指定包含其它的配置文件，可以在同一主机上多个Redis实例之间使用同一份配置文件，而同时各个实例又拥有自己的特定配置文件

  include /path/to/local.conf

# 10【掌握】Redis的持久化

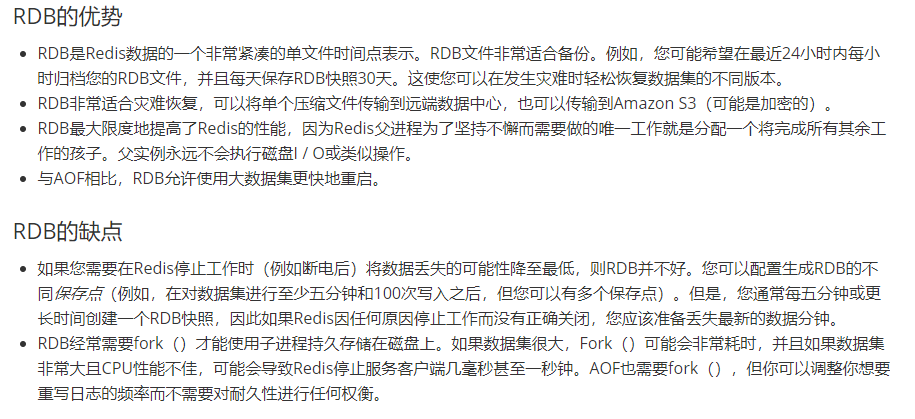
## 1，概述

<https://redis.io/topics/persistence>



## 2，RDB【Redis DataBase】

### 2.1，什么是RDB



save 900 1

save 300 10

save 60   10000

在指定的时间间隔内将内存中的数据集快照写入磁盘， 也就是行话讲的Snapshot快照，它恢复时是将快照文件直接读到内存里

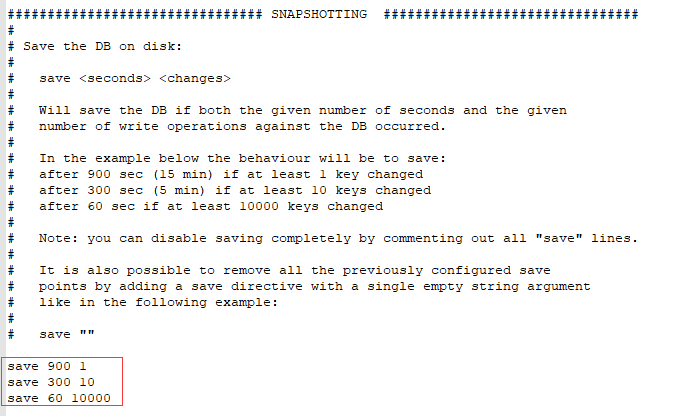
Redis会单独创建（fork）一个子进程来进行持久化，会先将数据写入到 一个临时文件中，待持久化过程都结束了，再用这个临时文件替换上次持久化好的文件。 整个过程中，主进程是不进行任何IO操作的，这就确保了极高的性能 如果需要进行大规模数据的恢复，且对于数据恢复的完整性不是非常敏感，那RDB方 式要比AOF方式更加的高效。RDB的缺点是最后一次持久化后的数据可能丢失。

### 2.2，什么是FORK

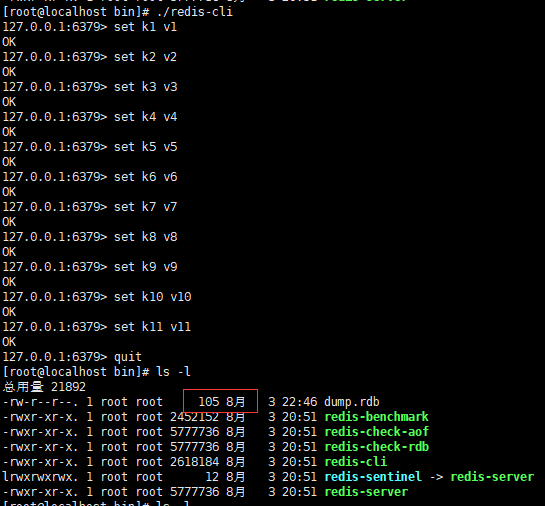
Fork的作用是复制一个与当前进程一样的进程。新进程的所有数据（变量、环境变量、程序计数器等） 数值都和原进程一致，但是是一个全新的进程，并作为原进程的子进程

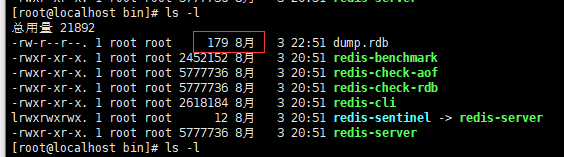
### 2.3，保存位置及配置位置

Rdb 保存的是dump.rdb文件



### 2.4，如何触发RDB快照



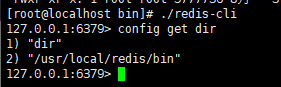


执行flushall命令，也会产生dump.rdb文件，但里面是空的，无意义

### 2.5，如何恢复数据

将备份文件 (dump.rdb) 移动到 redis 安装目录并启动服务即可

CONFIG GET dir获取目录



### 2.6，优点

适合大规模的数据恢复

对数据完整性和一致性要求不高

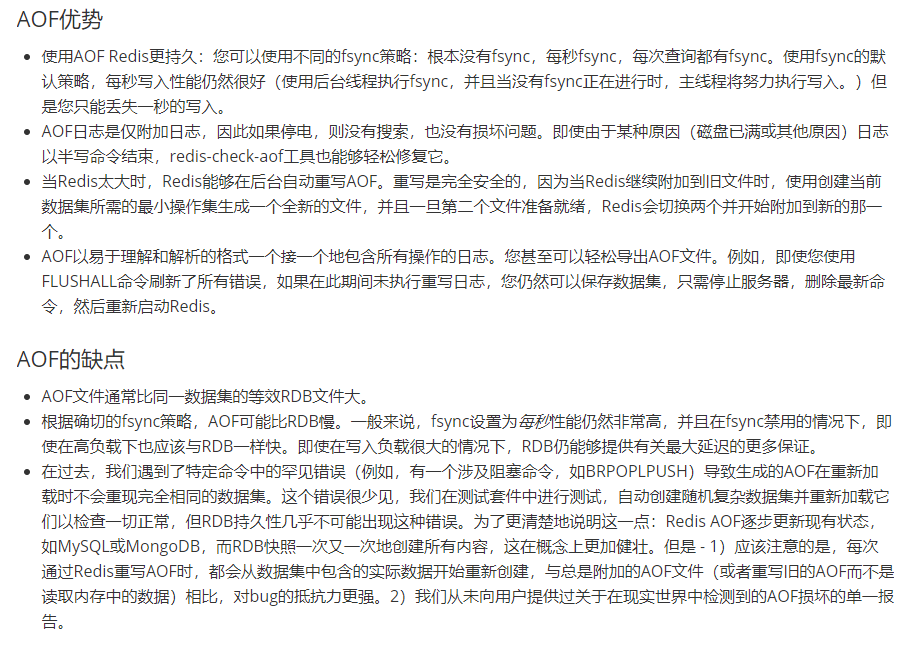
### 2.7，缺点

在一定间隔时间做一次备份，所以如果redis意外down掉的话，就 会丢失最后一次快照后的所有修改

Fork的时候，内存中的数据被克隆了一份，大致2倍的膨胀性需要考虑

## 3，AOF

### 3.1，概述



### 3.2，原理

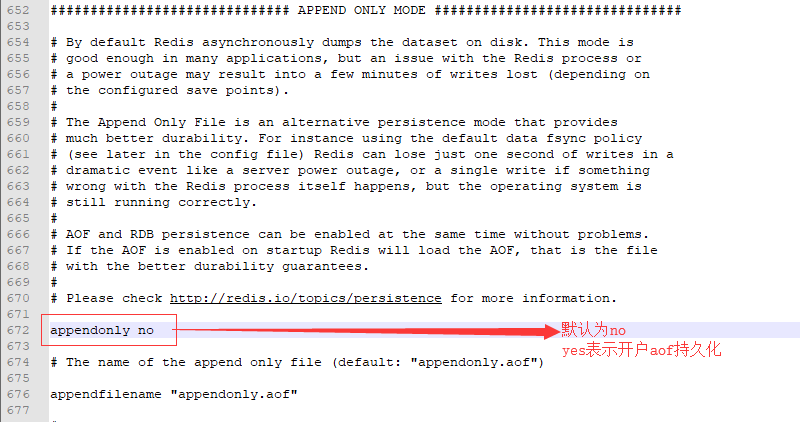
以日志的形式来记录每个写操作，将Redis执行过的所有写指令记录下来(读操作不记录)，

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重新构建数据，换言之，redis

重启的话就根据日志文件的内容将写指令从前到后执行一次以完成数据的恢复工作

### 3.3，保存位置及位置配置

Aof保存的是appendonly.aof文件



### 3.4，AOF启动/修复/恢复

**正常恢复**

  启动：设置Yes  修改默认的appendonly no，改为yes

  将有数据的aof文件复制一份保存到对应目录(config get dir)

  恢复：重启redis然后重新加载

**异常恢复**

  启动：设置Yes   修改默认的appendonly no，改为yes

  备份被写坏的AOF文件

  修复：  Redis-check-aof --fix进行修复

  恢复：重启redis然后重新加载

### 3.5，优势

  每修改同步：appendfsync always 同步持久化 每次发生数据变更会被立即记录到磁盘 性能较差但数据完整性比较好

  每秒同步：appendfsync everysec 异步操作，每秒记录 如果一秒内宕机，有数据丢失

  不同步：appendfsync no 从不同步

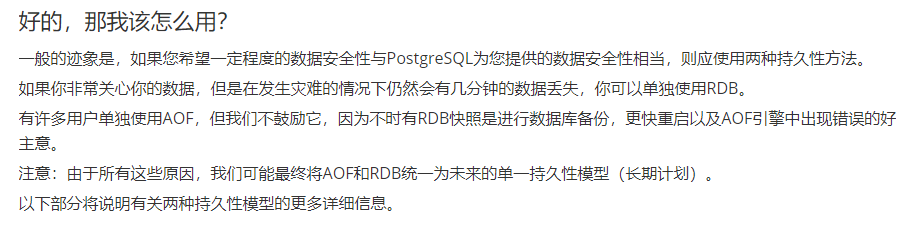
### 3.5，劣势

  相同数据集的数据而言aof文件要远大于rdb文件，恢复速度慢于rdb

  Aof运行效率要慢于rdb,每秒同步策略效率较好，不同步效率和rdb相同

## 4，说了那么多，我们选择哪一个呢

### 4.1，官方建议



**4.2，整理我们的理解及处理方式**

RDB持久化方式能够在指定的时间间隔能对你的数据进行快照存储

AOF持久化方式记录每次对服务器写的操作,当服务器重启的时候会重新执行这些 命令来恢复原始的数据,AOF命令以redis协议追加保存每次写的操作到文件末尾. Redis还能对AOF文件进行后台重写,使得AOF文件的体积不至于过大

**只做缓存**：如果你只希望你的数据在服务器运行的时候存在,你也可以不使用任何持久化方式.

**同时开启两种持久化方式**

  在这种情况下,当redis重启的时候会优先载入AOF文件来恢复原始的数据, 因为在通常情况下AOF文件保存的数据集要比RDB文件保存的数据集要完整.

  RDB的数据不实时，同时使用两者时服务器重启也只会找AOF文件。那要不要只使用AOF呢？ 作者建议不要，因为RDB更适合用于备份数据库(AOF在不断变化不好备份)， 快速重启，而且不会有AOF可能潜在的bug，留着作为一个万一的手段。

**性能建议**

因为RDB文件只用作后备用途，建议只在Slave上持久化RDB文件，而且只要15分钟备份一次就够了，只保留save 900 1这条规则。

如果Enalbe AOF，好处是在最恶劣情况下也只会丢失不超过两秒数据，启动脚本较简单只load自己的AOF文件就可以了。代价一是带来了持续的IO，二是AOF rewrite的最后将rewrite过程中产生的新数据写到新文件造成的阻塞几乎是不可避免的。只要硬盘许可，应该尽量减少AOF rewrite的频率，AOF重写的基础大小默认值64M太小了，可以设到5G以上。默认超过原大小100%大小时重写可以改到适当的数值。

如果不Enable AOF ，仅靠Master-Slave Replication 实现高可用性也可以。能省掉一大笔IO也减少了rewrite时带来的系统波动。代价是如果Master/Slave同时倒掉，会丢失十几分钟的数据，启动脚本也要比较两个Master/Slave中的RDB文件，载入较新的那个。新浪微博就选用了这种架构

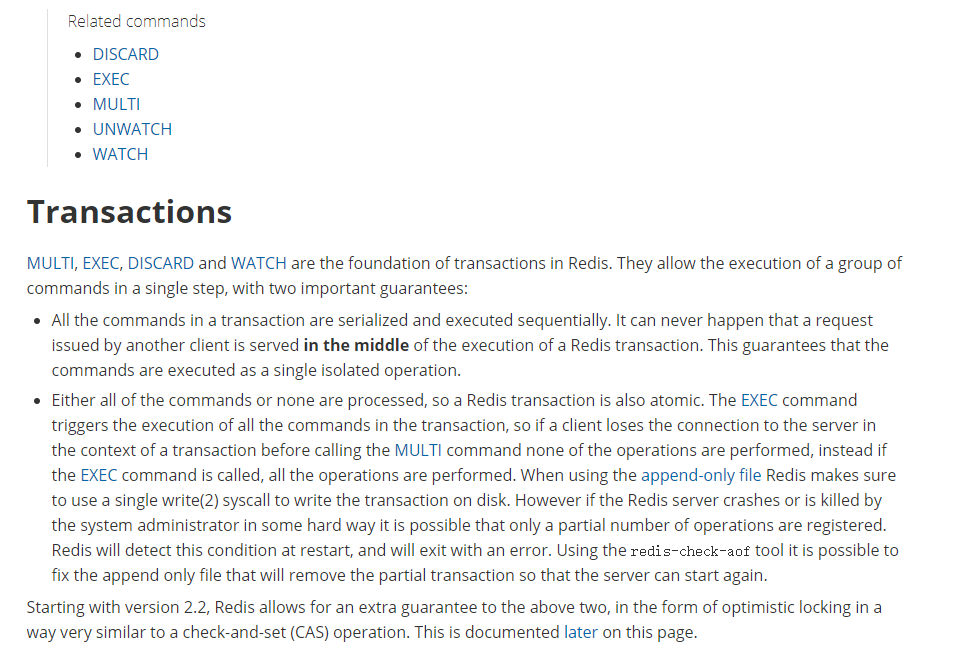
# 11【了解】Redis的事务

## 1，什么是Redis事务

  可以一次执行多个命令，本质是一组命令的集合。一个事务中的 所有命令都会序列化，按顺序地串行化执行而不会被其它命令插入，不许加塞

  官网说明

<https://redis.io/topics/transactions>



## 2，能为我们做什么

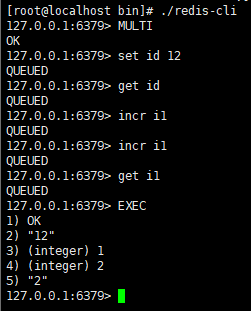
  一个队列中，一次性、顺序性、排它性的执行一系列命令

## 3，怎么使用呢？

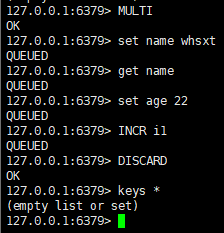
### 3.1，常用命令



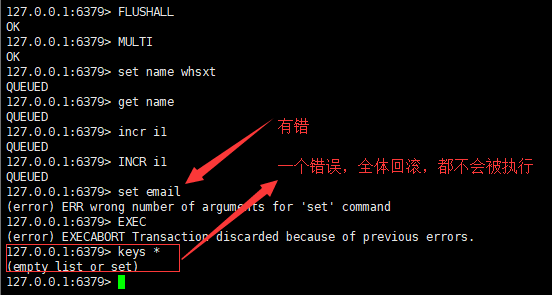
### 3.2， 情况1：正常执行



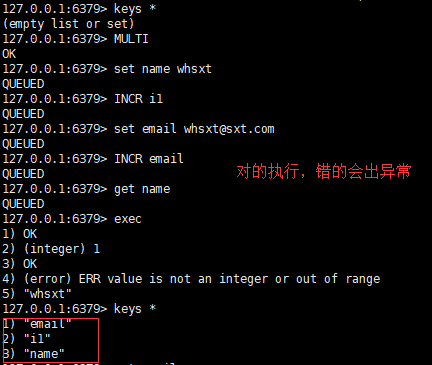
### 3.3，情况2：放弃事务



### 3.4， 情况3：全体连坐



### 3.5，情况4：冤头债主



### 3.6，情况5：watch监控

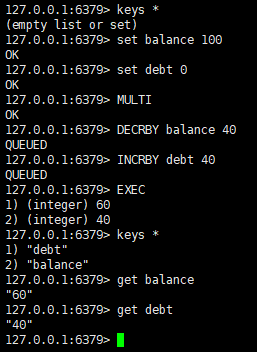
#### 3.6.1，悲观锁/乐观锁

**悲观锁(Pessimistic Lock),** 顾名思义，就是很悲观，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会block直到它拿到锁。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁

**乐观锁(Optimistic Lock)**, 顾名思义，就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制。乐观锁适用于多读的应用类型，这样可以提高吞吐量，

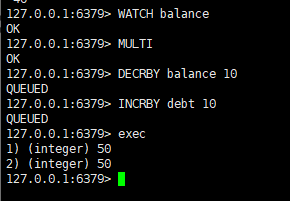
乐观锁策略:提交版本必须大于记录当前版本才能执行更新

**3.6.2，初始化信用卡可用余额和欠额**



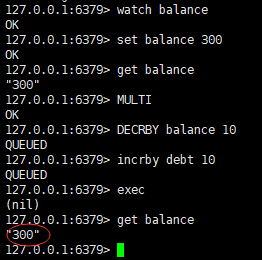
**3.6.2，无加塞篡改**

先监控再开启multi， 保证两笔金额变动在同一个事务内

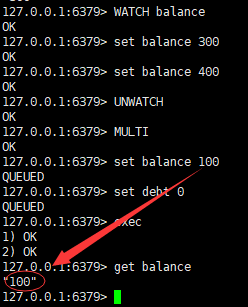


**3.6.2，有加塞篡改**

监控了key，如果key被修改了，后面一个事务的执行失效



**3.6.3，unwatch**



    一旦执行了exec之前加的监控锁都会被取消掉了

**3.6.4，小结**

      Watch指令，类似乐观锁，事务提交时，如果Key的值已被别的客户端改变， 比如某个list已被别的客户端push/pop过了，整个事务队列都不会被执行

      通过WATCH命令在事务执行之前监控了多个Keys，倘若在WATCH之后有任何Key的值发生了变化， EXEC命令执行的事务都将被放弃，同时返回Nullmulti-bulk应答以通知调用者事务执行失败

## 4，3阶段

**开启：**以MULTI开始一个事务

**入队：**将多个命令入队到事务中，接到这些命令并不会立即执行，而是放到等待执行的事务队列里面

**执行：**由EXEC命令触发事务

## 5，3特性

**单独的隔离操作：**事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。

**没有隔离级别的概念：**队列中的命令没有提交之前都不会实际的被执行，因为事务提交前任何指令都不会被实际执行， 也就不存在”事务内的查要看到事务里的更新，在事务外查询不能看到”这个让人万分头痛的问题

**不保证原子性：**redis同一个事务中如果有一条命令执行失败，其后的命令仍然会被执行，没有回滚

# 12【熟悉】Redis的复制(Master/Slave)

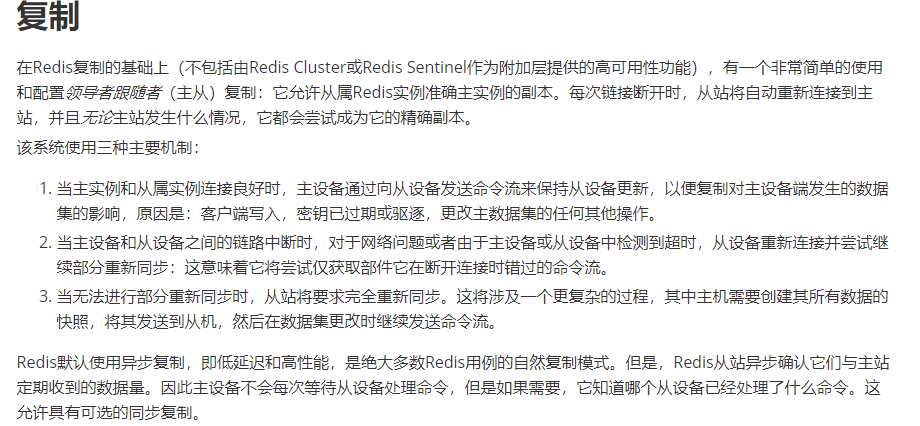
## 1，什么是复制

Redis 的读并发量太大怎么办？

单机版的Redis 挂掉怎么办？

需要写并发又要安全  在redis 3.0 后，官方发布了集群方案

### 1.1，官网说明



### 1.2，行话

行话：也就是我们所说的**主从复制**，主机数据更新后根据配置和策略，

自动同步到备机的master/slaver机制，Master以写为主，Slave以读为主

## 2，有什么作用

读写分离

容灾恢复

## 3，怎么使用

### 3.1，配从不配主

### 3.2，配置命令  【SLAVEOF 主库IP 主库端口】

每次与master断开之后，都需要重新连接，除非你配置进redis.conf文件

Info replication可以查看

### 3.3，详细操作

①拷贝多个redis.conf文件

②开启daemonize yes

③Pid文件名字

④指定端口

⑤Log文件名字

⑥Dump.rdb名字

### 3.4，一主二仆

演示问题

1 切入点问题？slave1、slave2是从头开始复制还是从切入点开始复制?比如从k4进来，那之前的123是否也可以复制

2 从机是否可以写？set可否？

3 主机shutdown后情况如何？从机是上位还是原地待命

4 主机又回来了后，主机新增记录，从机还能否顺利复制？

5 其中一台从机down后情况如何？依照原有它能跟上大部队吗？

### 3.5，薪火相传

上一个Slave可以是下一个slave的Master，Slave同样可以接收其他 slaves的连接和同步请求，那么该slave作为了链条中下一个的master, 可以有效减轻master的写压力

中途变更转向:会清除之前的数据，重新建立拷贝最新的

Slaveof 新主库IP 新主库端口

### 3.6，反客为主

SLAVEOF no one

  使当前数据库停止与其他数据库的同步，转成主数据库

## 4，复制的原理

Slave启动成功连接到master后会发送一个sync命令

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

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

增量复制：Master继续将新的所有收集到的修改命令依次传给slave,完成同步

但是只要是重新连接master,一次完全同步（全量复制)将被自动执行

## 5，哨兵模式

### 5.1，什么是哨兵模式

    反客为主的自动版，能够后台监控主机是否故障，如果故障了根据投票数自动将从库转换为主库

### 5.2，怎么玩(使用步骤)

    调整结构，6379带着80、81

    自定义的/myredis目录下新建sentinel.conf文件，名字绝不能错

    配置哨兵,填写内容

       sentinel monitor 被监控主机名字(自己起名字) 127.0.0.1 6379 1   如     sentinel monitor hos6379  127.0.0.1 6379 1

      上面最后一个数字1，表示主机挂掉后salve投票看让谁接替成为主机，得票数多少后成为主机

    启动哨兵   ./redis-sentinel /root/master-slaver/sentinel.conf

    正常主从演示

    原有的master挂了

    投票新选

    重新主从继续开工,info replication查查看

    问题：如果之前的master重启回来，会不会双master冲突？

### 5.3，一组sentinel能同时监控多个Master

## 6，复制的缺点

由于所有的写操作都是先在Master上操作，然后同步更新到Slave上，所以从Master同步到Slave机器有一定的延迟，当系统很繁忙的时候，延迟问题会更加严重，Slave机器数量的增加也会使这个问题更加严重。

# 13【掌握】高可用高并发集群配置

## 1，中心化和去中心化

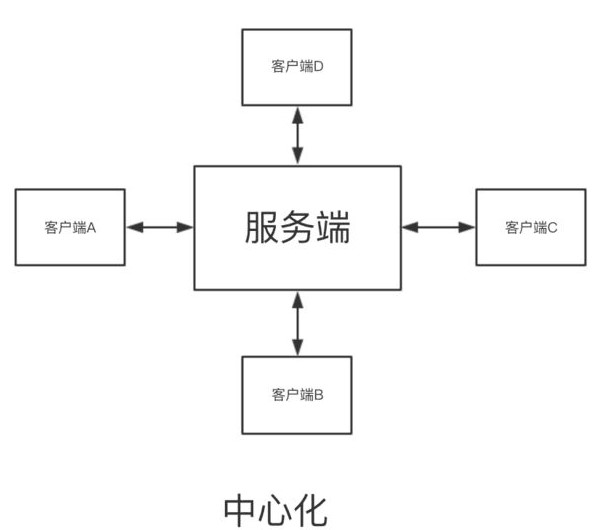
### 1.1，中心化

意思是所有的节点都要有一个主节点

缺点：中心挂了，服务就挂了

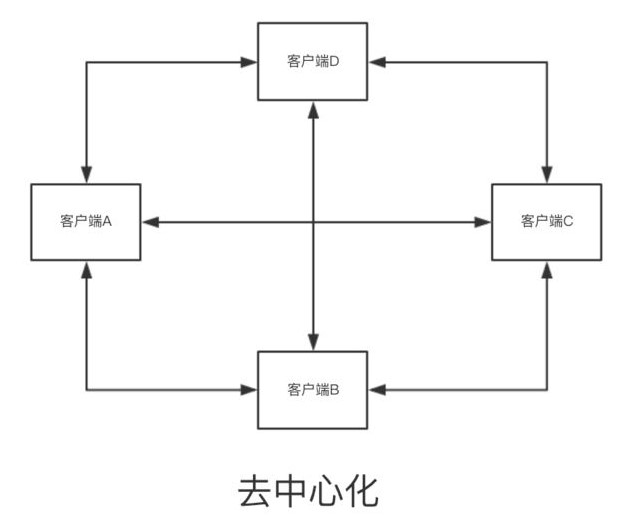
            中心处理数据的能力有限，不能把节点性能发挥到最大

特点：就是一个路由作用



### 1.2，去中心化

特点：去掉路由，我自己来路由



以上通俗的就是

中心化：几个经过认证的嘉宾在‘讲话’，所有其他人在听。

去中心化：每个人都可以‘讲话’，每个人都可以选择听或者讲。

## 2，Redis集群的执行流程分析

### 2.1，哈希槽说明

Redis 集群中内置了 16384 个哈希槽，当需要在 Redis 集群中放置一个 key-value时，redis 先对 key 使用 crc16 算法算出一个结果，然后把结果对 16384 求余数，这样每个 key 都会对应一个编号在 0-16383 之间的哈希槽，redis 会根据节点数量大致均等的将哈希槽映射到不同的节点。

当你往Redis Cluster中加入一个Key时，会根据crc16(key) mod 16384计算这个key应该分布到哪个hash slot中，一个hash slot中会有很多key和value。你可以理解成表的分区，使用单节点时的redis时只有一个表，所有的key都放在这个表里；改用Redis Cluster以后会自动为你生成16384个分区表，你insert数据时会根据上面的简单算法来决定你的key应该存在哪个分区，每个分区里有很多key。

### 2.2，执行流程分析

假如redis集群里面能存放90个key，那么redis集群把90key平分到3个主机

redis对每个主机里面30个存储位置都编号，当应用连接到主机1上面时，应该发送一个写的命令

主机使用crc16算出槽号

如果槽号在1-30 可以直接操作主机1

如果槽号在31-60那么redis会转发到主机2

如果应该再发一个命令set age 22

那么主机2使用crc16再算槽号再转发

## 3，Redis集群的搭建

### 3.1，文档

<http://redis.cn/topics/cluster-tutorial.html>

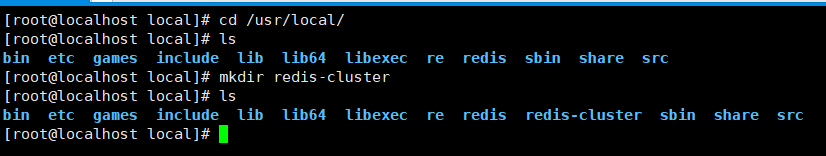
### 3.2，原理：去中心化

### 3.3，集群规则

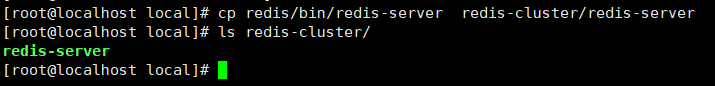
|  |  |  |
| --- | --- | --- |
| 机器编号 | ip | port |
| 1 | 192.168.120.129 | 7000 |
| 2 | 192.168.120.129 | 7001 |
| 3 | 192.168.120.129 | 7002 |
| 4 | 192.168.120.129 | 7003 |
| 5 | 192.168.120.129 | 7004 |
| 6 | 192.168.120.129 | 7005 |

### 3.4，搭建过程

#### 3.4.1 新建文件夹

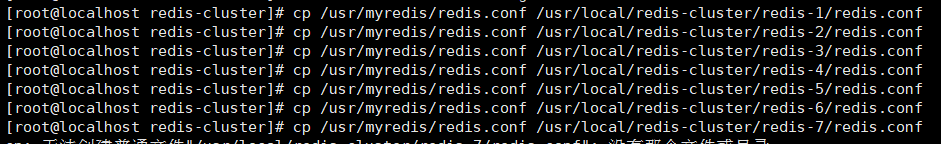


#### 3.4.2 准备一个服务端程序



#### 3.4.3准备6个redis的配置文件





Redis-1

bind 0.0.0.0 69行

port 7000 92行

daemonize yes 136行

# 打开aof 持久化

appendonly yes 672行

# 开启集群

cluster-enabled yes 814行

# 集群的配置文件,该文件自动生成

cluster-config-file nodes-7000.conf 822行

# 集群的超时时间

cluster-node-timeout 5000 828行

------------------------------------

Redis-2

daemonize yes

bind 0.0.0.0

port 7001

# 打开aof 持久化

appendonly yes

# 开启集群

cluster-enabled yes

# 集群的配置文件,该文件自动生成

cluster-config-file nodes-7001.conf

# 集群的超时时间

cluster-node-timeout 5000

-------------------------------------

Redis-3

daemonize yes

bind 0.0.0.0

port 7002

# 打开aof 持久化

appendonly yes

# 开启集群

cluster-enabled yes

# 集群的配置文件,该文件自动生成

cluster-config-file nodes-7002.conf

# 集群的超时时间

cluster-node-timeout 5000

------------------------------------

Redis-4

daemonize yes

bind 0.0.0.0

port 7003

# 打开aof 持久化

appendonly yes

# 开启集群

cluster-enabled yes

# 集群的配置文件,该文件自动生成

cluster-config-file nodes-7004.conf

# 集群的超时时间

cluster-node-timeout 5000

-------------------------------------

Redis-5

daemonize yes

bind 0.0.0.0

port 7004

# 打开aof 持久化

appendonly yes

# 开启集群

cluster-enabled yes

# 集群的配置文件,该文件自动生成

cluster-config-file nodes-7005.conf

# 集群的超时时间

cluster-node-timeout 5000

​

--------------------------------------

Redis-6

daemonize yes

bind 0.0.0.0

port 7005

# 打开aof 持久化

appendonly yes

# 开启集群

cluster-enabled yes

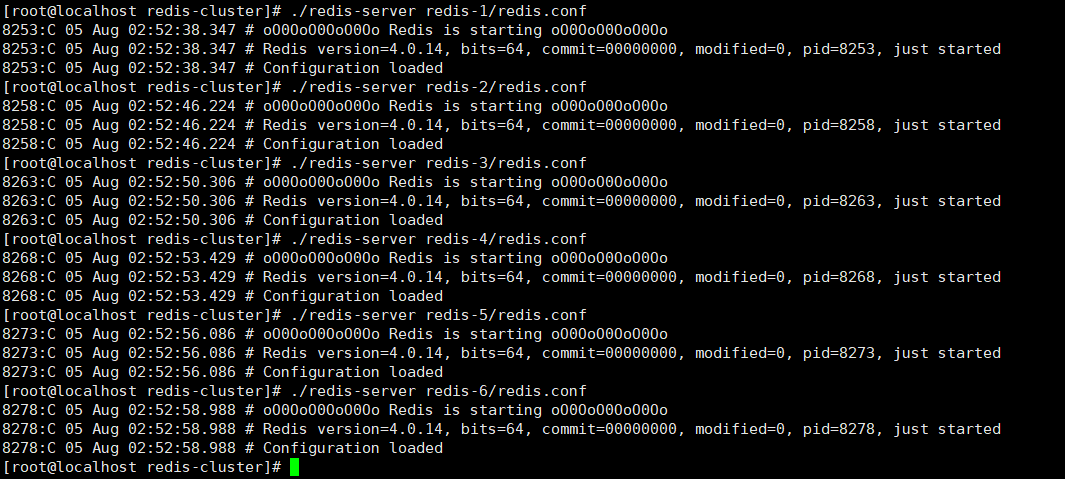
# 集群的配置文件,该文件自动生成

cluster-config-file nodes-7006.conf

# 集群的超时时间

cluster-node-timeout 5000

#### 3.4.3同时启动所有的redis



#### 3.4.4 使用脚本创建集群（分配槽）

 找到集群脚本，在src/src/redis-trib.rb  要安装Ruby的环境【不推荐

#### 3.4.5 使用docker 下载redis-trib的镜像运行【推荐】

安装Docker

yum install docker

启动docker

systemctl start docker

A: 下载镜像

docker pull inem0o/redis-trib

docker run -it --net host inem0o/redis-trib create --replicas 1

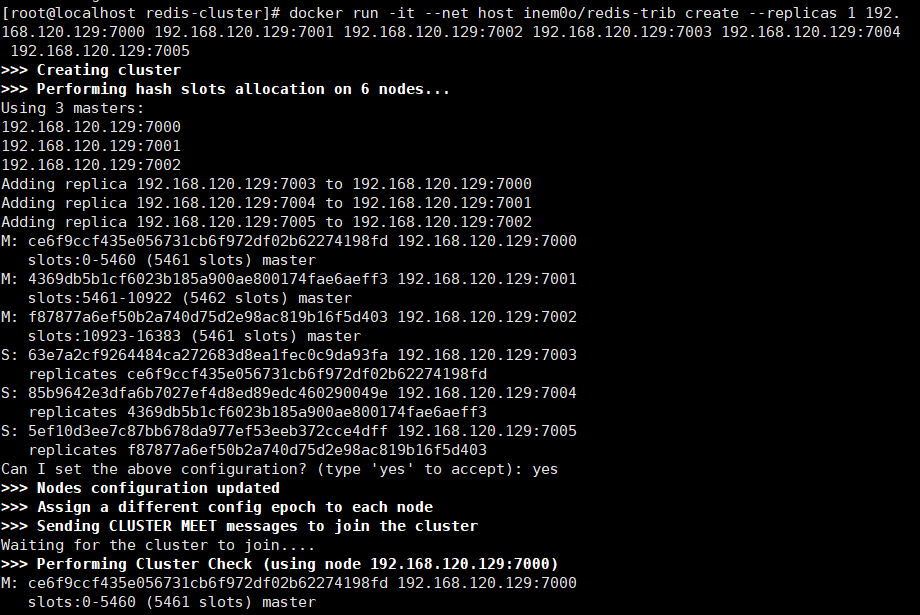
192.168.120.129:7000 192.168.120.129:7001

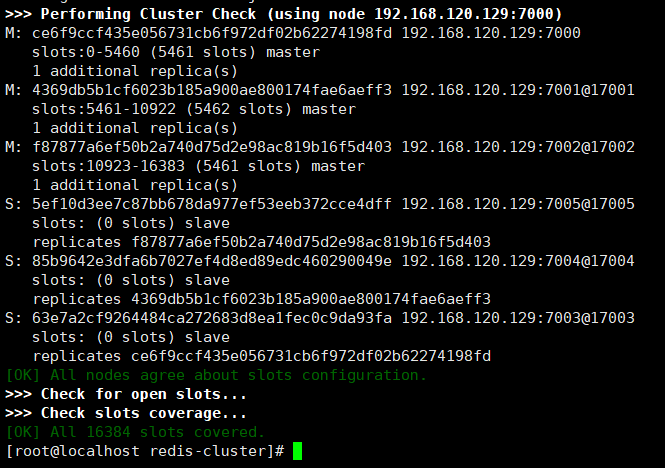
192.168.120.129:7002 192.168.120.129:7003

192.168.120.129:7004 192.168.120.129:7005

-it是为了可以输入

--net host 是为了上docker容器能连接上本地的宿主机

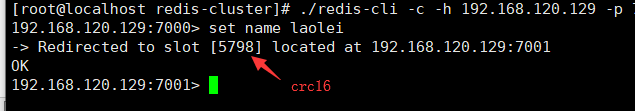


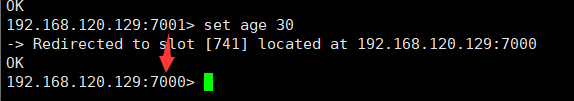


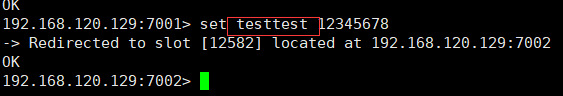
#### 3.4.6测试集群环境



-c 表示连接集群





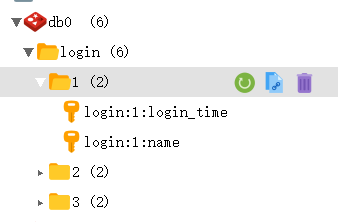


到此集群搭建完成

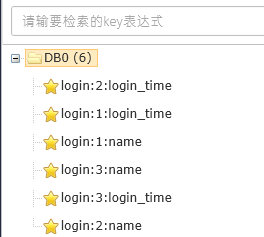
# 14【掌握】Redis客户端使用

## 1， 工具说明

    1，使用【可以看到层级关系】



    2，使用



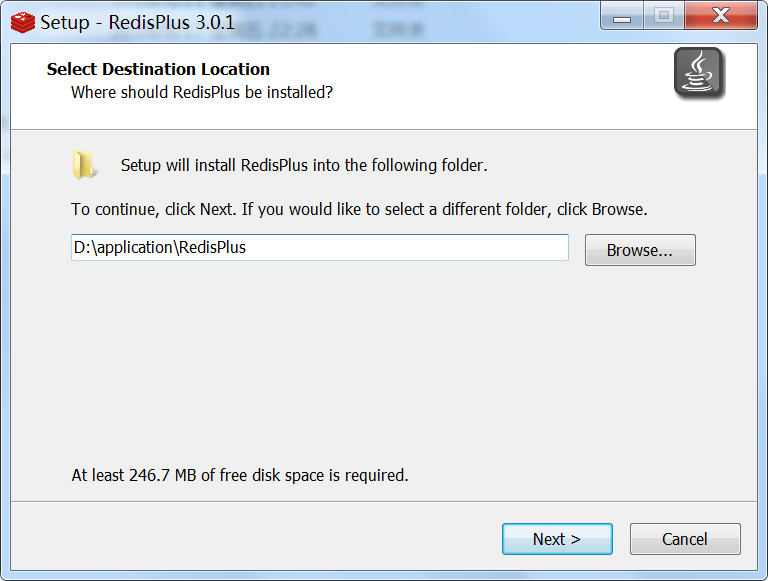
这个没有层级关系

## 2， 下载客户端redis plus

链接：https://pan.baidu.com/s/1hPBErS--3zOZv6EXL2PEsA

提取码：048c

## 3，安装





## 3，连接

1，关闭linux防火墙

Centos6

查看防火墙状态：

[root@centos6 ~]# service iptables status

iptables：未运行防火墙。

开启防火墙：

[root@centos6 ~]# service iptables start

关闭防火墙：

[root@centos6 ~]# service iptables stop

centos7

查看防火墙状态：

firewall-cmd --state

关闭防火墙

systemctl stop firewalld.service

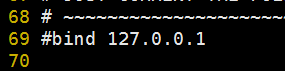
开启防火墙

systemctl start firewalld.service

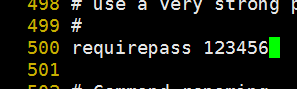
禁用防火墙

systemctl disable firewalld.service

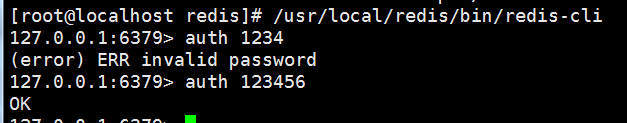
2，注释redis.conf里面的bind 127.0.0.1



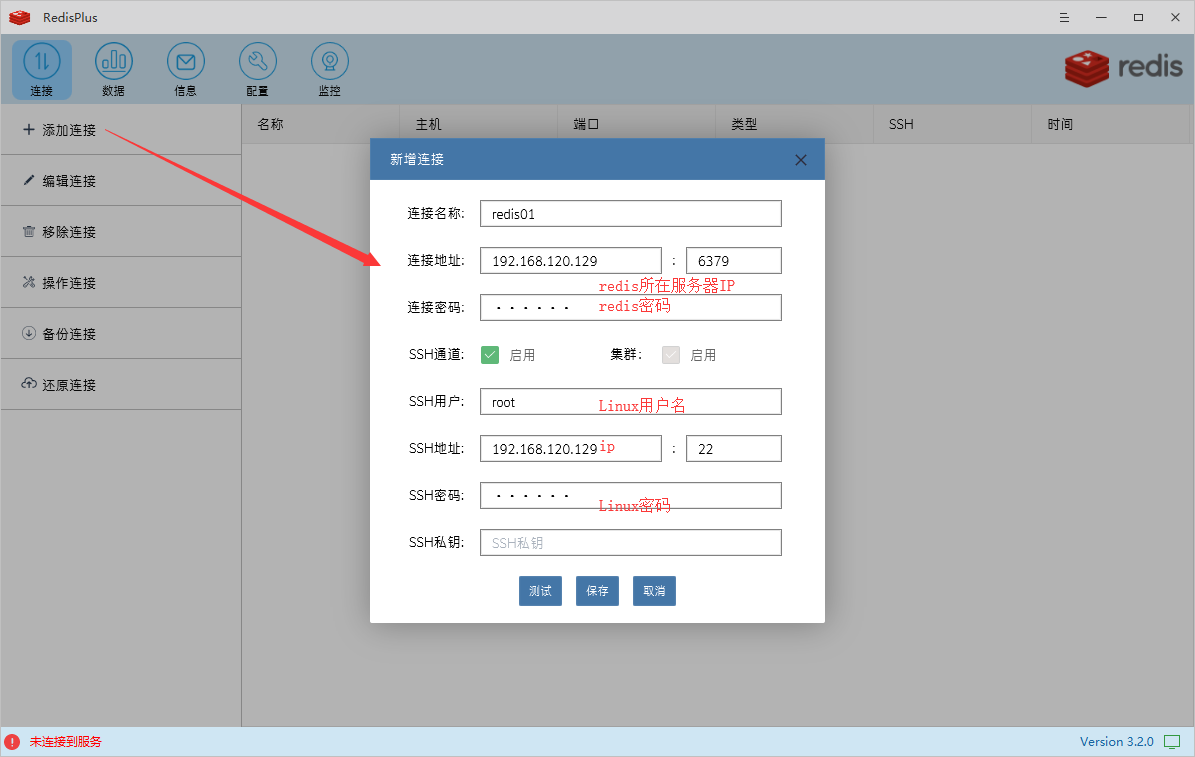
3，修改redis.conf里面的密码



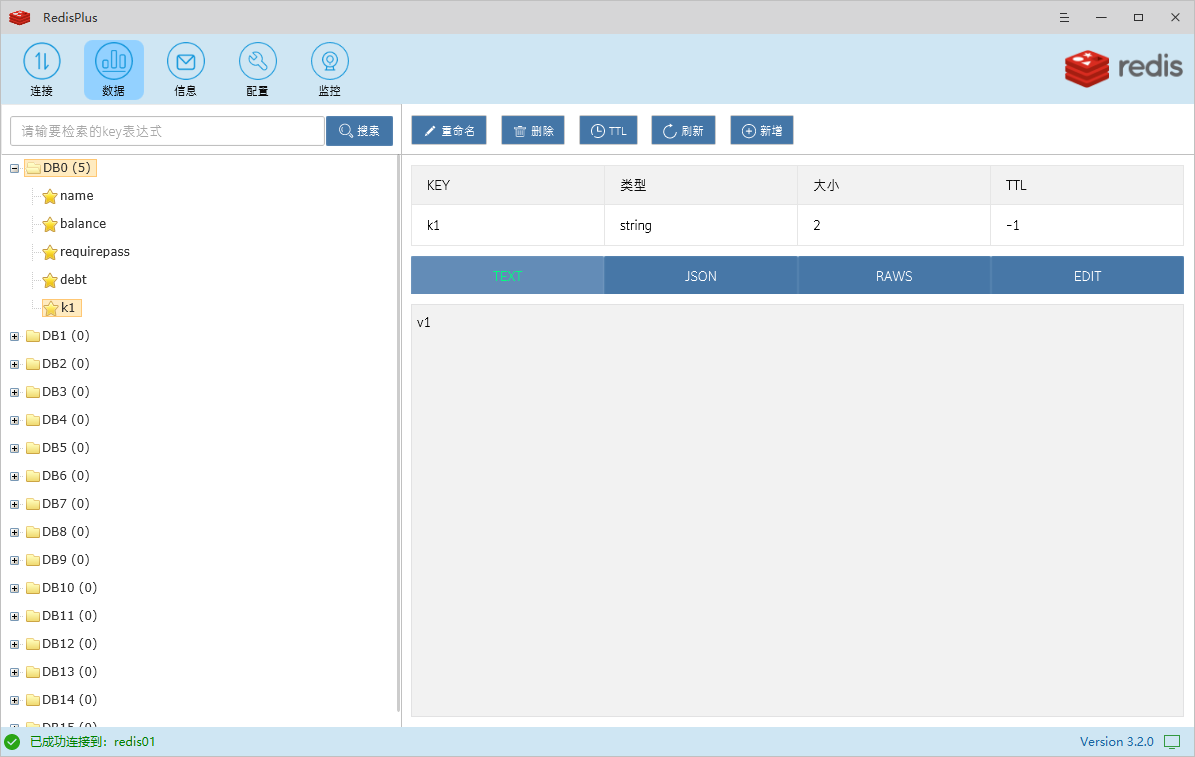
4，验证密码是否成功



5，连接



6，连接成功



# 15【掌握】java连接redis

## 1，Jedis所需要的jar包依赖

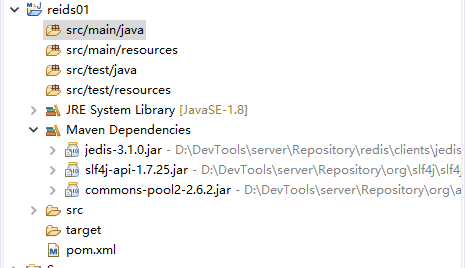
<dependency>

<groupId>redis.clients</groupId>

<artifactId>jedis</artifactId>

<version>3.1.0</version>

</dependency>



## 2，Jedis常用操作

### 2.1，测试连通性

public class TestRedisPing {

public static void main(String[] args) {

//创建连接

Jedis jedis=new Jedis("192.168.120.129", 6379);

//设置密码 如果没有密码可以不设置

jedis.auth("123456");

//调用 ping方法

String ping = jedis.ping();

//输出PONG

System.out.println(ping);

jedis.close();

}

}

### 2.2，常规操作【自学】

public class TestReidsCommon {

​

public static void main(String[] args) {

// 创建连接

Jedis jedis = new Jedis("192.168.120.129", 6379);

// 设置密码 如果没有密码可以不设置

jedis.auth("123456");

// key

Set<String> keys = jedis.keys("\*");

for (Iterator<String> iterator = keys.iterator(); iterator.hasNext();) {

String key = (String) iterator.next();

System.out.println(key);

}

System.out.println("jedis.exists====>" + jedis.exists("k2"));

System.out.println(jedis.ttl("k1"));

// String

// jedis.append("k1","myreids");

System.out.println(jedis.get("k1"));

jedis.set("k4", "k4\_redis");

System.out.println("----------------------------------------");

jedis.mset("str1", "v1", "str2", "v2", "str3", "v3");

System.out.println(jedis.mget("str1", "str2", "str3"));

// list

System.out.println("----------------------------------------");

// jedis.lpush("mylist","v1","v2","v3","v4","v5");

List<String> list = jedis.lrange("mylist", 0, -1);

for (String element : list) {

System.out.println(element);

}

// set

jedis.sadd("orders", "jd001");

jedis.sadd("orders", "jd002");

jedis.sadd("orders", "jd003");

Set<String> set1 = jedis.smembers("orders");

for (Iterator<String> iterator = set1.iterator(); iterator.hasNext();) {

String string = (String) iterator.next();

System.out.println(string);

}

jedis.srem("orders", "jd002");

System.out.println(jedis.smembers("orders").size());

// hash

jedis.hset("hash1", "userName", "lisi");

System.out.println(jedis.hget("hash1", "userName"));

Map<String, String> map = new HashMap<String, String>();

map.put("telphone", "15902738715");

map.put("address", "whsxt");

map.put("email", "leijharvin@163.com");

jedis.hmset("hash2", map);

List<String> result = jedis.hmget("hash2", "telphone", "email");

for (String element : result) {

System.out.println(element);

}

// zset

jedis.zadd("zset01", 60d, "v1");

jedis.zadd("zset01", 70d, "v2");

jedis.zadd("zset01", 80d, "v3");

jedis.zadd("zset01", 90d, "v4");

​

Set<String> s1 = jedis.zrange("zset01", 0, -1);

for (Iterator<String> iterator = s1.iterator(); iterator.hasNext();) {

String string = (String) iterator.next();

System.out.println(string);

}

jedis.close();

}

}

### 2.3，事务提交

常规操作

public class TestReidsTranaction {

​

public static void main(String[] args) {

// 创建连接

Jedis jedis = new Jedis("192.168.120.129", 6379);

// 设置密码 如果没有密码可以不设置

jedis.auth("123456");

//监控key，如果该动了事务就被放弃

/\*3

jedis.watch("serialNum");

jedis.set("serialNum","s#####################");

jedis.unwatch();\*/

Transaction transaction = jedis.multi();//被当作一个命令进行执行

Response<String> response = transaction.get("serialNum");

transaction.set("serialNum","s002");

response = transaction.get("serialNum");

transaction.lpush("list3","a");

transaction.lpush("list3","b");

transaction.lpush("list3","c");

transaction.exec();

//2 transaction.discard();

System.out.println("serialNum\*\*\*\*\*\*\*\*\*\*\*"+response.get());

jedis.close();

​

}

​

}

​

加锁操作

public class TestReidsTranactionLock {

​

public boolean transMethod() {

// 创建连接

Jedis jedis = new Jedis("192.168.120.129", 6379);

// 设置密码 如果没有密码可以不设置

jedis.auth("123456");

int balance;// 可用余额

int debt;// 欠额

int amtToSubtract = 10;// 实刷额度

​

jedis.watch("balance");

// jedis.set("balance","5");//此句不该出现，讲课方便。模拟其他程序已经修改了该条目

balance = Integer.parseInt(jedis.get("balance"));

if (balance < amtToSubtract) {

jedis.unwatch();

System.out.println("modify");

jedis.close();

return false;

} else {

System.out.println("\*\*\*\*\*\*\*\*\*\*\*transaction");

Transaction transaction = jedis.multi();

transaction.decrBy("balance", amtToSubtract);

transaction.incrBy("debt", amtToSubtract);

transaction.exec();

balance = Integer.parseInt(jedis.get("balance"));

debt = Integer.parseInt(jedis.get("debt"));

System.out.println("\*\*\*\*\*\*\*" + balance);

System.out.println("\*\*\*\*\*\*\*" + debt);

jedis.close();

return true;

}

}

​

/\*\*

\* 通俗点讲，watch命令就是标记一个键，如果标记了一个键， 在提交事务前如果该键被别人修改过，那事务就会失败，这种情况通常可以在程序中 重新再尝试一次。

\* 首先标记了键balance，然后检查余额是否足够，不足就取消标记，并不做扣减； 足够的话，就启动事务进行更新操作，

\* 如果在此期间键balance被其它人修改， 那在提交事务（执行exec）时就会报错， 程序中通常可以捕获这类错误再重新执行一次，直到成功。

\*/

public static void main(String[] args) {

TestReidsTranactionLock test = new TestReidsTranactionLock();

boolean retValue = test.transMethod();

System.out.println("main retValue-------: " + retValue);

}

​

}

### 2.4，主从复制

public class TestReidsMS {

​

public static void main(String[] args) throws InterruptedException {

// 创建连接

Jedis jedis\_M = new Jedis("192.168.120.129", 6379);

Jedis jedis\_S = new Jedis("192.168.120.129", 6380);

​

jedis\_S.slaveof("127.0.0.1", 6379);

​

jedis\_M.set("k6", "v6");

​

Thread.sleep(500);

System.out.println(jedis\_S.get("k6"));

jedis\_M.close();

jedis\_S.close();

}

​

}

## 3，JedisPool

### 3.1，为什么要使用JedisPool

  1，获取Jedis实例需要从JedisPool中获取

  2，用完Jedis实例需要返还给JedisPool

  3，如果Jedis在使用过程中出错，则也需要还给JedisPool

### 3.2，案例见代码

    JedisPoolUtil

public class JedisPoolUtil {

​

private static volatile JedisPool jedisPool = null;// 被volatile修饰的变量不会被本地线程缓存，对该变量的读写都是直接操作共享内存。

​

private JedisPoolUtil() {

}

​

public static JedisPool getJedisPoolInstance() {

if (null == jedisPool) {

synchronized (JedisPoolUtil.class) {

if (null == jedisPool) {

JedisPoolConfig poolConfig = new JedisPoolConfig();

poolConfig.setMaxTotal(1000);

poolConfig.setMaxIdle(32);

poolConfig.setMaxWaitMillis(100 \* 1000);

poolConfig.setTestOnBorrow(true);

​

jedisPool = new JedisPool(poolConfig, "192.168.120.129", 6379);

}

}

}

return jedisPool;

}

​

public static void release(JedisPool jedisPool, Jedis jedis) {

if (null != jedis) {

Jedis jedis2 = null;

try {

jedis2 = jedisPool.getResource();

} finally {

jedis2.close();

}

}

}

​

}

使用

public class TestJedisPool {

public static void main(String[] args) {

JedisPool jedisPool = JedisPoolUtil.getJedisPoolInstance();

Jedis jedis = null;

​

try {

jedis = jedisPool.getResource();

jedis.set("k18", "v183");

​

} catch (Exception e) {

e.printStackTrace();

} finally {

JedisPoolUtil.release(jedisPool, jedis);

}

}

​

}

​

  配置总结all

JedisPool的配置参数大部分是由JedisPoolConfig的对应项来赋值的。

maxTotal/maxActive：控制一个pool可分配多少个jedis实例，通过pool.getResource()来获取；如果赋值为-1，则表示不限制；如果pool已经分配了maxActive个jedis实例，则此时pool的状态为exhausted。

maxIdle：控制一个pool最多有多少个状态为idle(空闲)的jedis实例；

whenExhaustedAction：表示当pool中的jedis实例都被allocated完时，pool要采取的操作；默认有三种。

WHEN\_EXHAUSTED\_FAIL --> 表示无jedis实例时，直接抛出NoSuchElementException；

WHEN\_EXHAUSTED\_BLOCK --> 则表示阻塞住，或者达到maxWait时抛出JedisConnectionException；

WHEN\_EXHAUSTED\_GROW --> 则表示新建一个jedis实例，也就说设置的maxActive无用；

setMaxWaitMillis/maxWait：表示当borrow一个jedis实例时，最大的等待时间，如果超过等待时间，则直接抛JedisConnectionException；

testOnBorrow：获得一个jedis实例的时候是否检查连接可用性（ping()）；如果为true，则得到的jedis实例均是可用的；

testOnReturn：return 一个jedis实例给pool时，是否检查连接可用性（ping()）；

testWhileIdle：如果为true，表示有一个idle object evitor线程对idle object进行扫描，如果validate失败，此object会被从pool中drop掉；这一项只有在timeBetweenEvictionRunsMillis大于0时才有意义；

timeBetweenEvictionRunsMillis：表示idle object evitor两次扫描之间要sleep的毫秒数；

numTestsPerEvictionRun：表示idle object evitor每次扫描的最多的对象数；

minEvictableIdleTimeMillis：表示一个对象至少停留在idle状态的最短时间，然后才能被idle object evitor扫描并驱逐；这一项只有在timeBetweenEvictionRunsMillis大于0时才有意义；

softMinEvictableIdleTimeMillis：在minEvictableIdleTimeMillis基础上，加入了至少minIdle个对象已经在pool里面了。如果为-1，evicted不会根据idle time驱逐任何对象。如果minEvictableIdleTimeMillis>0，则此项设置无意义，且只有在timeBetweenEvictionRunsMillis大于0时才有意义；

lifo：borrowObject返回对象时，是采用DEFAULT\_LIFO（last in first out，即类似cache的最频繁使用队列），如果为False，则表示FIFO队列；

==================================================================================================================

其中JedisPoolConfig对一些参数的默认设置如下：

testWhileIdle=true

minEvictableIdleTimeMills=60000

timeBetweenEvictionRunsMillis=30000

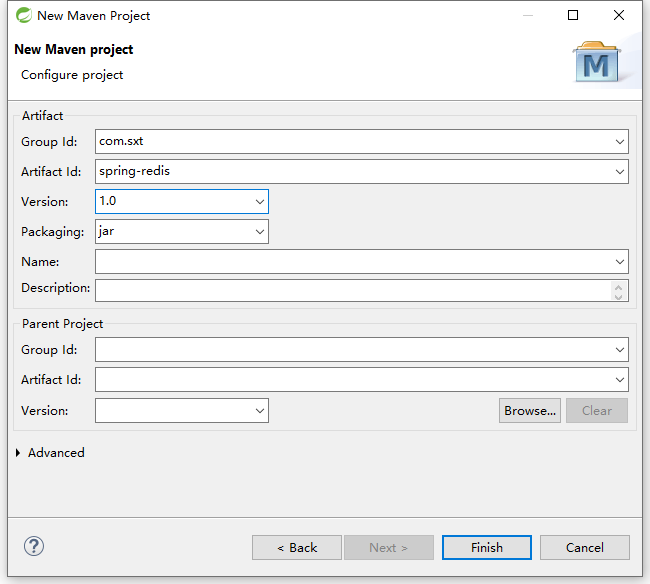
numTestsPerEvictionRun=-1

​

# 16【掌握】spring里面使用

## 1，spring集成redis

### 1.1，创建项目



### 1.2，修改pom.xml

<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>com.sxt</groupId>

<artifactId>spring-redis</artifactId>

<version>1.0</version>

​

<!-- jar包版本声明 -->

<properties>

<spring.version>4.3.24.RELEASE</spring.version>

<jedis.version>3.1.0</jedis.version>

</properties>

<!-- 声明需要的依赖的具体的资源 -->

<dependencies>

<!-- 导入spring -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aspects</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aop</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-jdbc</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>redis.clients</groupId>

<artifactId>jedis</artifactId>

<version>${jedis.version}</version>

</dependency>

</dependencies>

</project>

### 1.3，搭建spring的环境

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">

<bean id="poolConfig" class="redis.clients.jedis.JedisPoolConfig">

<property name="maxIdle" value="20"></property>

<property name="maxTotal" value="25"></property>

<property name="minIdle" value="10"></property>

</bean>

<bean id="jedisPool" class="redis.clients.jedis.JedisPool">

<constructor-arg name="poolConfig" ref="poolConfig"></constructor-arg>

<constructor-arg name="host" value="192.168.120.130"></constructor-arg>

<constructor-arg name="port" value="6379"></constructor-arg>

<constructor-arg name="password" value="123456"></constructor-arg>

<constructor-arg name="timeout" value="5000"></constructor-arg>

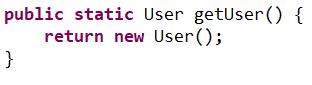
</bean>

</beans>

<bean id=”user” class=”com.sxt.domain.User”>

<property name=”” ref/value=””>

</bean>

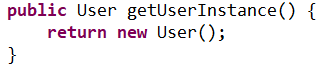


    修改配置文件获取对象的方式



3 用实例工厂创建UserFactory

    创建不是静态方法，返回类对象

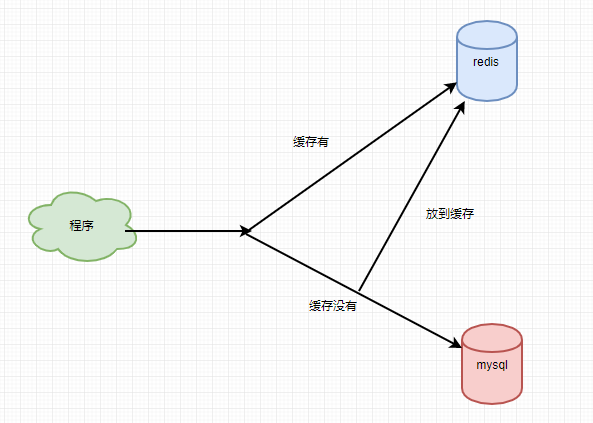


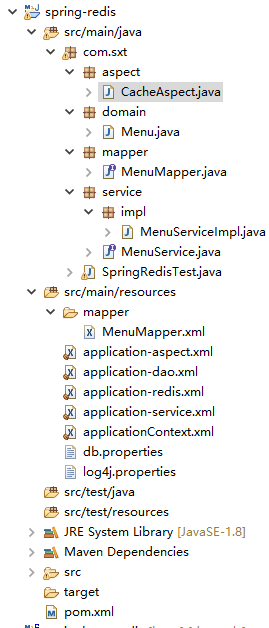
    修改配置文件获取对象



## 2，实现菜单数据的缓存

### 2.1，简单原理图





### 2.2，修改pom.xml

<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>com.sxt</groupId>

<artifactId>spring-redis</artifactId>

<version>1.0</version>

​

<!-- jar包版本声明 -->

<properties>

<mybatis.version>3.5.2</mybatis.version>

<mybatis-spring.version>2.0.2</mybatis-spring.version>

<spring.version>4.3.24.RELEASE</spring.version>

<druid.version>1.0.18</druid.version>

<mysql.version>8.0.17</mysql.version>

<fastjson.version>1.2.59</fastjson.version>

<!-- 注意只能使用2.0以下的版本 -->

<log4j.version>1.2.17</log4j.version>

<jedis.version>3.1.0</jedis.version>

</properties>

<!-- 声明需要的依赖的具体的资源 -->

<dependencies>

<!-- mybatis -->

<dependency>

<groupId>org.mybatis</groupId>

<artifactId>mybatis</artifactId>

<version>${mybatis.version}</version>

</dependency>

<!-- mybatis-spring -->

<dependency>

<groupId>org.mybatis</groupId>

<artifactId>mybatis-spring</artifactId>

<version>${mybatis-spring.version}</version>

</dependency>

<!-- 导入spring -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aspects</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aop</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context-support</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-jdbc</artifactId>

<version>${spring.version}</version>

</dependency>

<!-- mysql数据库驱动 -->

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>${mysql.version}</version>

</dependency>

<!-- json -->

<dependency>

<groupId>com.alibaba</groupId>

<artifactId>fastjson</artifactId>

<version>${fastjson.version}</version>

</dependency>

<!-- log4j -->

<dependency>

<groupId>log4j</groupId>

<artifactId>log4j</artifactId>

<version>${log4j.version}</version>

</dependency>

<dependency>

<groupId>redis.clients</groupId>

<artifactId>jedis</artifactId>

<version>${jedis.version}</version>

</dependency>

</dependencies>

​

​

</project>

### 2.3，创建Menu

public class Menu {

private Integer id;

private Integer pid;

private String title;

private String href;

private Integer spread;

private String target;

private String icon;

private Integer available;

//get set方法

}

### 2.4，创建MenuMapper

public interface MenuMapper {

/\*\*

\* 查询所有菜单

\*/

List<Menu> queryAllMenu();

}

### 2.5，创建MenuMapper.xml

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace="com.sxt.mapper.MenuMapper">

<resultMap id="BaseResultMap" type="com.sxt.domain.Menu">

<id column="id" jdbcType="INTEGER" property="id" />

<result column="pid" jdbcType="INTEGER" property="pid" />

<result column="title" jdbcType="VARCHAR" property="title" />

<result column="href" jdbcType="VARCHAR" property="href" />

<result column="spread" jdbcType="INTEGER" property="spread" />

<result column="target" jdbcType="VARCHAR" property="target" />

<result column="icon" jdbcType="VARCHAR" property="icon" />

<result column="available" jdbcType="INTEGER"

property="available" />

</resultMap>

<sql id="Base\_Column\_List">

id, pid, title, href, spread, target, icon, available

</sql>

<!-- 查询所有菜单 -->

<select id="queryAllMenu" resultMap="BaseResultMap">

select

<include refid="Base\_Column\_List" />

from sys\_menu

</select>

</mapper>

### 2.6，创建MenuService

/\*\*

\* 菜单管理的服务接口

\* @author LJH

\*

\*/

public interface MenuService {

​

/\*\*

\* 查询所有菜单返回

\* List<Menu>

\*/

public List<Menu> queryAllMenuForList();

}

### 2.7，创建MenuServiceImpl

@Service

public class MenuServiceImpl implements MenuService {

@Autowired

private MenuMapper menuMapper;

​

@Override

public List<Menu> queryAllMenuForList() {

return menuMapper.queryAllMenu();

}

}

### 2.8，application-dao.xml

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

xmlns:aop="http://www.springframework.org/schema/aop"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd

http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd">

​

<!-- 引入db.properties -->

<context:property-placeholder

location="classpath:db.properties" system-properties-mode="FALLBACK" />

​

<!-- 声明dataSource -->

<bean id="dataSource"

class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<!-- 注入连接属性 -->

<property name="driverClassName" value="${driverClassName}"></property>

<property name="url" value="${url}"></property>

<property name="username" value="${username}"></property>

<property name="password" value="${password}"></property>

</bean>

<!-- 声明sessionFactory 并注入mybatis.cfg.xml -->

<bean id="sqlSessionFactory"

class="org.mybatis.spring.SqlSessionFactoryBean">

<!-- 注入数据源 -->

<property name="dataSource" ref="dataSource"></property>

<!-- 注入mapper.xml -->

<property name="mapperLocations">

<array>

<value>classpath:mapper/\*Mapper.xml</value>

</array>

</property>

</bean>

​

<!-- 扫描mapper接口 -->

<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">

<!-- 注入mapper接口所在的包 注意多个包的情况的配置 -->

<property name="basePackage">

<value>

com.sxt.mapper

</value>

</property>

<!-- 注入sqlSessionFactory -->

<property name="sqlSessionFactoryBeanName"

value="sqlSessionFactory"></property>

</bean>

</beans>

​

### 2.9，application-service.xml

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

xmlns:aop="http://www.springframework.org/schema/aop"

xmlns:tx="http://www.springframework.org/schema/tx"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd

http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd

http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx.xsd">

<context:component-scan base-package="com.sxt.service.impl"></context:component-scan>

<!-- 1,声明事务管理器 -->

<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

<property name="dataSource" ref="dataSource"></property>

</bean>

<!-- 启动注解事务 -->

<!-- <tx:annotation-driven/> -->

<!-- 2,声明事务的传播特性 也就是通知 -->

<tx:advice id="advise" transaction-manager="transactionManager">

<tx:attributes>

<!-- 以add开头的方法名需要事务 -->

<tx:method name="add\*" propagation="REQUIRED"/>

<tx:method name="save\*" propagation="REQUIRED"/>

<tx:method name="update\*" propagation="REQUIRED"/>

<tx:method name="delete\*" propagation="REQUIRED"/>

<tx:method name="change\*" propagation="REQUIRED"/>

<tx:method name="reset\*" propagation="REQUIRED"/>

<tx:method name="get\*" read-only="true"/>

<tx:method name="load\*" read-only="true"/>

<tx:method name="\*" read-only="true"/>

</tx:attributes>

</tx:advice>

<!-- 3进行AOP织入 -->

<aop:config>

<!-- 声明切面 -->

<aop:pointcut expression="execution(\* com.sxt.service.impl.\*.\*(..))" id="pc"/>

<!-- 织入 -->

<aop:advisor advice-ref="advise" pointcut-ref="pc"/>

</aop:config>

</beans>

​

### 2.10，application-redis.xml

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">

<bean id="poolConfig" class="redis.clients.jedis.JedisPoolConfig">

<property name="maxIdle" value="20"></property>

<property name="maxTotal" value="25"></property>

<property name="minIdle" value="10"></property>

</bean>

<bean id="jedisPool" class="redis.clients.jedis.JedisPool">

<constructor-arg name="poolConfig" ref="poolConfig"></constructor-arg>

<constructor-arg name="host" value="192.168.120.130"></constructor-arg>

<constructor-arg name="port" value="6379"></constructor-arg>

<constructor-arg name="password" value="123456"></constructor-arg>

<constructor-arg name="timeout" value="5000"></constructor-arg>

</bean>

​

</beans>

​

### 2.11，applicationContext.xml

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">

<import resource="classpath:application-dao.xml"/>

<import resource="classpath:application-service.xml"/>

<import resource="classpath:application-redis.xml"/>

</beans>

### 2.12，log4j.properties

# Global logging configuration

log4j.rootLogger=DEBUG, stdout

# MyBatis logging configuration...

log4j.logger.org.mybatis.example.BlogMapper=TRACE

# Console output...

log4j.appender.stdout=org.apache.log4j.ConsoleAppender

log4j.appender.stdout.layout=org.apache.log4j.PatternLayout

log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n

### 2.13，db.properties

driverClassName=com.mysql.cj.jdbc.Driver

url=jdbc:mysql://127.0.0.1:3306/carrent?useUnicode=true&characterEncoding=UTF-8&serverTimezone=UTC

username=root

password=123456

### 2.14，CacheAspect

@Component

@Aspect

@EnableAspectJAutoProxy

public class CacheAspect {

@Autowired

private JedisPool jedisPool;

private static final String ALL\_MENU\_LABEL = "alll-menu-data";

@Pointcut("execution(\* com.sxt.service.impl.MenuServiceImpl.queryAllMenuForList())")

public void menuQueryPc() {

}

@Around(value="menuQueryPc()")

public Object cache(ProceedingJoinPoint point) {

Jedis jedis = jedisPool.getResource();

if(jedis.exists(ALL\_MENU\_LABEL)) {

String menuJson = jedis.get(ALL\_MENU\_LABEL);

List<Menu> menu = JSON.parseArray(menuJson, Menu.class);

return menu ;

}

Object result = null ;

try {

System.out.println("执行真实方法的调用");

result = point.proceed(point.getArgs()); // 在此实现了真实方法的调用

jedis.set(ALL\_MENU\_LABEL,JSON.toJSONString(result));

} catch (Throwable e) {

e.printStackTrace();

}

return result;

}

}

### 2.14，application-aspect.xml

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-4.3.xsd">

​

<context:component-scan base-package="com.sxt.aspect"></context:component-scan>

​

</beans>

​

### 2.15，applicationContext.xml

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">

<import resource="classpath:application-dao.xml"/>

<import resource="classpath:application-service.xml"/>

<import resource="classpath:application-redis.xml"/>

<import resource="classpath:application-aspect.xml"/>

</beans>

​

### 2.16，打断点测试

public class SpringRedisTest {

public static void main(String[] args) {

ApplicationContext context=new ClassPathXmlApplicationContext("classpath:applicationContext.xml");

MenuService menuService=context.getBean(MenuService.class);

//接下来的操作就和java里面一样的了

List<Menu> list = menuService.queryAllMenuForList();

for (Menu menu : list) {

System.out.println(menu);

}

}

​

}

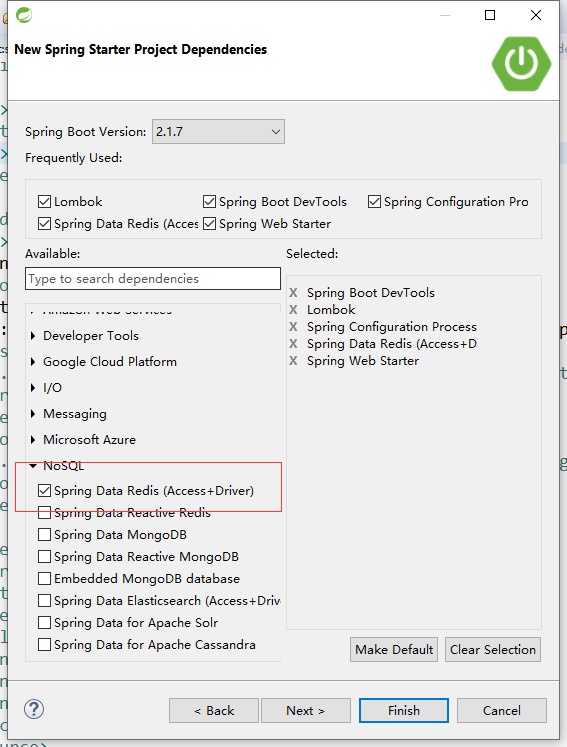
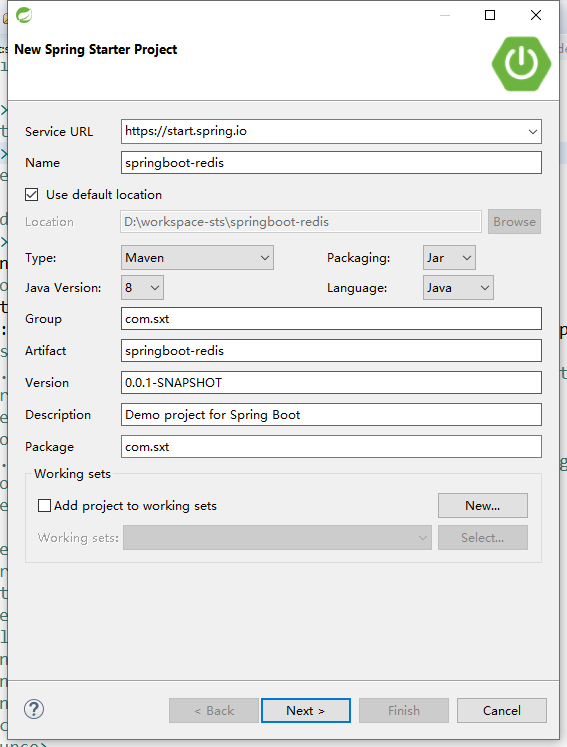
# 17【掌握】spring+redis实现秒杀

1. 页面 倒计时
2. 倒计时结束，用户可以点击按钮抢购
3. 只要用户点击，相当于一个线程到后台服务器 (一次启动1000线程去抢购)
4. 把商品初始化到redis list
   1. Lpop
   2. rpop

|  |
| --- |
| **public** **class** TestRedisMKill {  **private** **static** **final** String ***PRODUCTID*** = "productid";  **public** **static** **void** main(String[] args) {    ApplicationContext context=**new** ClassPathXmlApplicationContext("classpath:applicationContext.xml");  //向redis初始化商品  JedisPool jedisPool = context.getBean(JedisPool.**class**);  Jedis jedis = jedisPool.getResource();  jedis.flushAll();  **for** (**int** i = 1; i <=100; i++) {  jedis.lpush(***PRODUCTID***, "商品ID: "+i);  }  jedis.close();    //模拟线程抢购  **for** (**int** i = 1; i <=1000; i++) {  **new** Thread(**new** Runnable() {  @Override  **public** **void** run() {  Jedis jedis2 = jedisPool.getResource();  String lpop = jedis2.lpop(***PRODUCTID***);  **if**(**null**==lpop) {  System.***out***.println(Thread.*currentThread*().getName()+":抢购失败");  }**else** {  System.***out***.println(Thread.*currentThread*().getName()+":抢购成功 商品ID:"+lpop);  //向MQ发送一个消息 把商品ID和用户ID传到后台 启动一个线程去处理抢购成功这后的业务  }  jedis2.close();  }  },"张三"+i).start();  }  }      } |

# 18【掌握】springboot中使用redis

## 1，创建项目并选择依赖



## 3，修改配置文件yml

#redis的配置

spring:

redis:

host: 192.168.120.130

port: 6379

password: 123456

jedis:

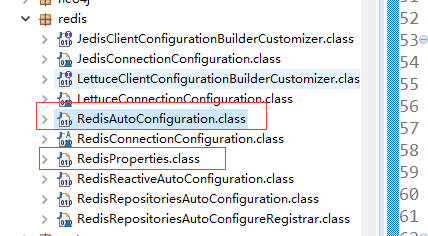
pool:

max-idle: 20

max-active: 25

min-idle: 10

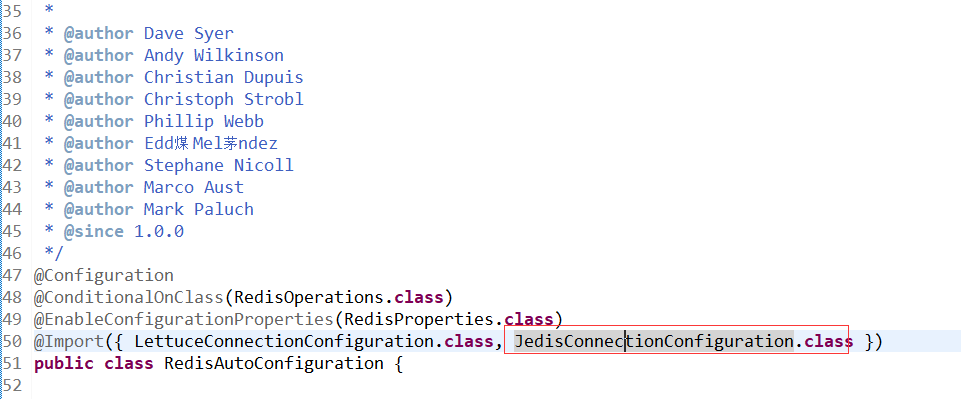
## 2，spring boot 如何加载默认的对象

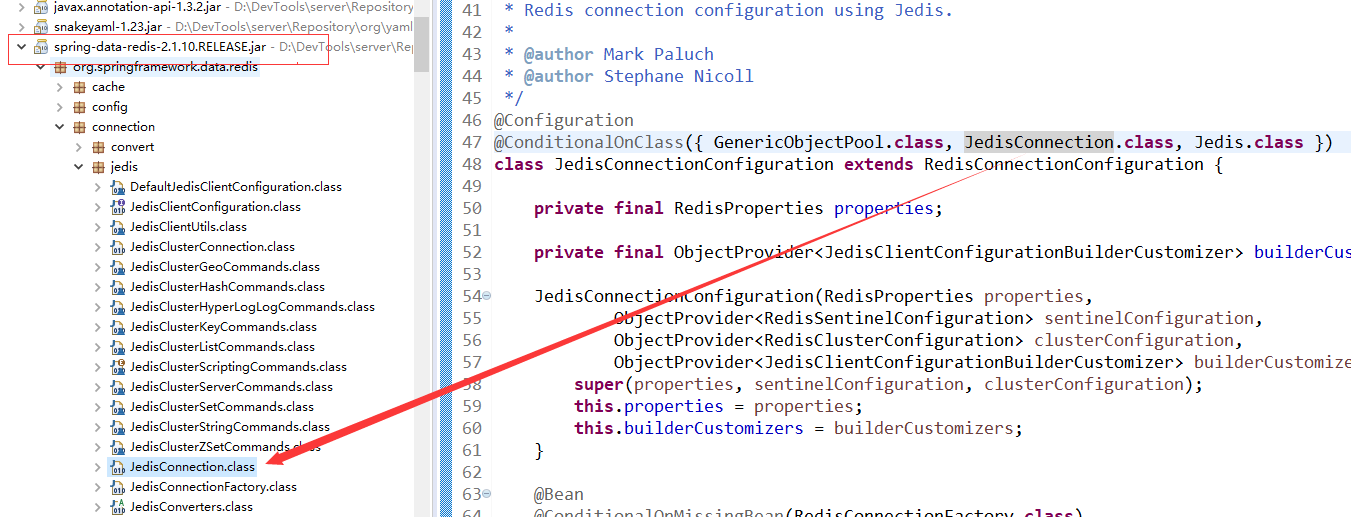


RedisAutoCongiguration创建对象

RedisProperties读取配置文件

为什么会配置成功呢？





可以看出是由springdata提供的访问对象

## 4，StringRedisTemplate (使用最多) 操作Redis

extends RedisTemplate<String,String>

|  |
| --- |
| /\*\*  \* 测试  \* @author LJH  \*  \*/  @RunWith(SpringRunner.class)  @SpringBootTest  public class SpringbootRedisApplicationTests {  @Autowired  private StringRedisTemplate redisTemplate;  /\*\*  \* redis数据类型为String的操作  \*/  @Test  public void testString() {  // 操作String类型  ValueOperations<String, String> opsValue = redisTemplate.opsForValue();  // 给redis 里面set 一个key  opsValue.set("boot", "spring-boot"); // k -v 都是String  // 从redis 里面获取key  String value = opsValue.get("boot");  System.out.println(value);  // 从redis 里面或多个key  List<String> asList = Arrays.asList("boot", "alll-menu-data");  List<String> mulitValues = opsValue.multiGet(asList);  System.out.println(mulitValues);  // redis的自动增长  Long increment = opsValue.increment("boot-incr", 2);// delta 可以+ 任意的数（步长）  System.out.println(increment);  }  @Test  public void testHash() {  HashOperations<String, Object, Object> opsForHash = redisTemplate.opsForHash();  // hset  opsForHash.put("object-1", "name", "sxt"); // 后面的2 个参数都是object,但是只支持String 类型  opsForHash.put("object-1", "age", "27"); // 后面的2 个参数都是object,但是只支持String 类型  opsForHash.put("object-1", "sex", "man"); // 后面的2 个参数都是object,但是只支持String 类型  Object value = opsForHash.get("object-1", "sex");  System.out.println(value);  // 取多个值  List<Object> multiGet = opsForHash.multiGet("object-1", Arrays.asList("name", "sex"));  System.out.println(multiGet);  }  @Test  public void testZset() {  ZSetOperations<String, String> opsForZSet = redisTemplate.opsForZSet();  // 放到zset集合里面  opsForZSet.add("lol", "sxt", 2500);  opsForZSet.add("lol", "lz", 0);  opsForZSet.add("lol", "ln", 1400);  opsForZSet.add("lol", "ll", -10);  opsForZSet.add("lol", "lt", 2700);  Set<String> rangeAsc = opsForZSet.range("lol", 0, 2); // 通过排序取值 ll lz ln  System.out.println(rangeAsc);  Set<String> reverseRange = opsForZSet.reverseRange("lol", 0, 2);// lt lz ln  System.out.println(reverseRange);  Set<TypedTuple<String>> tuples = new HashSet<ZSetOperations.TypedTuple<String>>();  tuples.add(new DefaultTypedTuple<String>("sxt", 1000.00));  tuples.add(new DefaultTypedTuple<String>("lv", 1200.00));  tuples.add(new DefaultTypedTuple<String>("lz", 2900.00));  tuples.add(new DefaultTypedTuple<String>("lt", 100.00));  // 若redis 存在该key ，则需要数据类型相同，不然报错  opsForZSet.add("dnf", tuples);  }  } |

## 5， RedisTemplate<Object,Object>（扩展String类型）操作Redis

若不设置序列化规则，它将使用JDK自动的序列化将对象转换为字节，存到Redis 里面

它可以存在对象到redis里面

如果对象没有序列化，那么默认使用的JDK的序列化方式



/\*\*

\* 测试

\* @author LJH

\*

\*/

@RunWith(SpringRunner.class)

@SpringBootTest

public class SpringbootRedisTemplateTests {

@Autowired

private RedisTemplate<Object, Object> redisTemplate ; // 因为创建RedisTemplate 没有使用泛型信息来创建，泛型 本质还是Object，只不过泛型能自动推断并强转

​

​

@Test

public void testString() {

redisTemplate.setKeySerializer(new StringRedisSerializer()); // key的序列化使用String 类型来完成 因为key 很多时候都是一个字符串

redisTemplate.setValueSerializer(new GenericJackson2JsonRedisSerializer()); // 优先没有泛型的，因为有泛型后，它支持的数据类型就定了

ValueOperations<Object, Object> valueOperations = redisTemplate.opsForValue();

// valueOperations.set("boot-redis", "boot-value"); //对象->字符串 json

User user = new User(1, "laolei", "xx.jpg", "78414842@qq.com");

// KEY : com.sxt.domain.User:1

// com.fasterxml.jackson.databind.JsonSerializer 没有依赖jackson 之前大家可能使用spring-boot-web，这里面会自动依赖

valueOperations.set(User.class.getName()+":"+user.getId(), user);

// 若该对象的强转转换，则redis 内部会使用JackSon 的工具将字符串-> 转换为java 对象 ，那jackson 转换为对象时，需要一个对象的类型 ，其实它已经自动对象的类型了"@class": "com.sxt.domain.User",

User object = (User)valueOperations.get(User.class.getName()+":"+user.getId());

System.out.println(object.getName()+":"+object.getIcon());

}

/\*\*

\* hash

\*/

@Test

public void testHash() {

redisTemplate.setKeySerializer(new StringRedisSerializer()); // key的序列化使用String 类型来完成 因为key 很多时候都是一个字符串

redisTemplate.setHashKeySerializer(new StringRedisSerializer());

redisTemplate.setHashValueSerializer(new StringRedisSerializer()); // 若都是string 则和StringRedisTempalte一样了

HashOperations<Object, Object, Object> opsForHash = redisTemplate.opsForHash();

opsForHash.put("redis-hash", "prop1", "value");

}

}

## 6，集群的额外操作

@Test

public void testCluster() {

ClusterOperations<Object, Object> opsForCluster = redisTemplate.opsForCluster();

//关闭集群的7000端口的主机

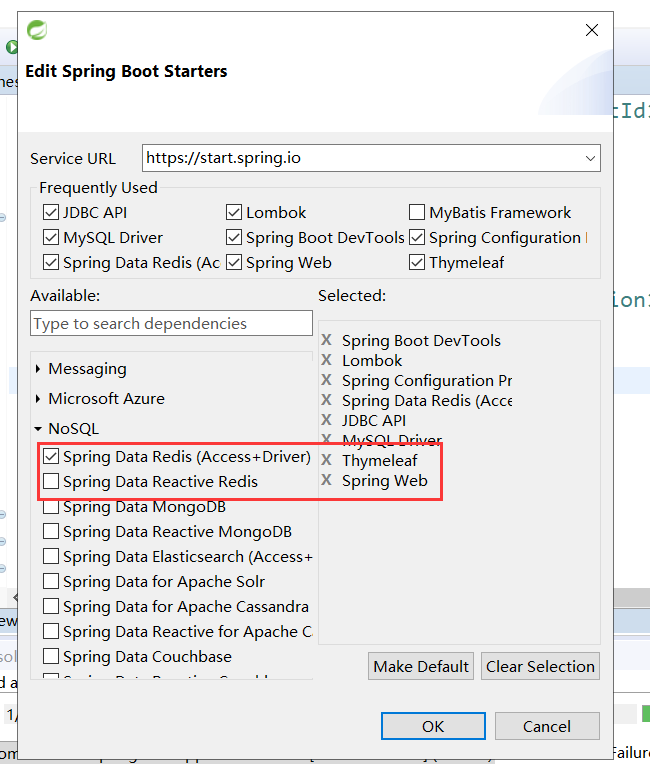
opsForCluster.shutdown(new RedisClusterNode("192.168.120.130", 7000));

}

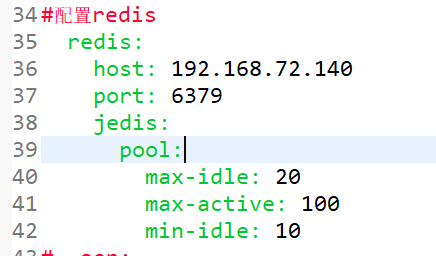
# 18【掌握】springboot使用redis做缓存

## 修改仓库管理项目把前的缓存改成redis

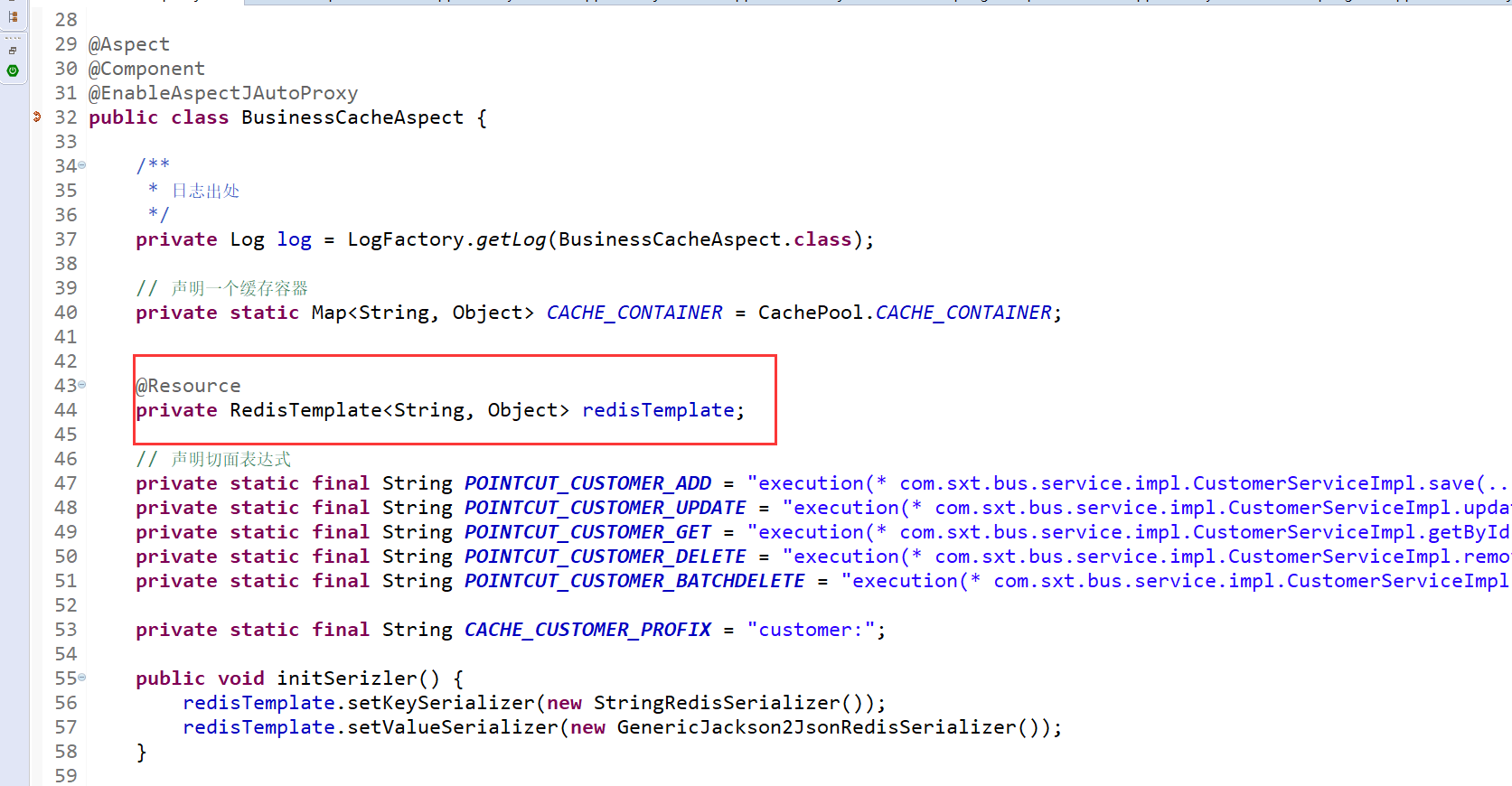
### 添加依赖

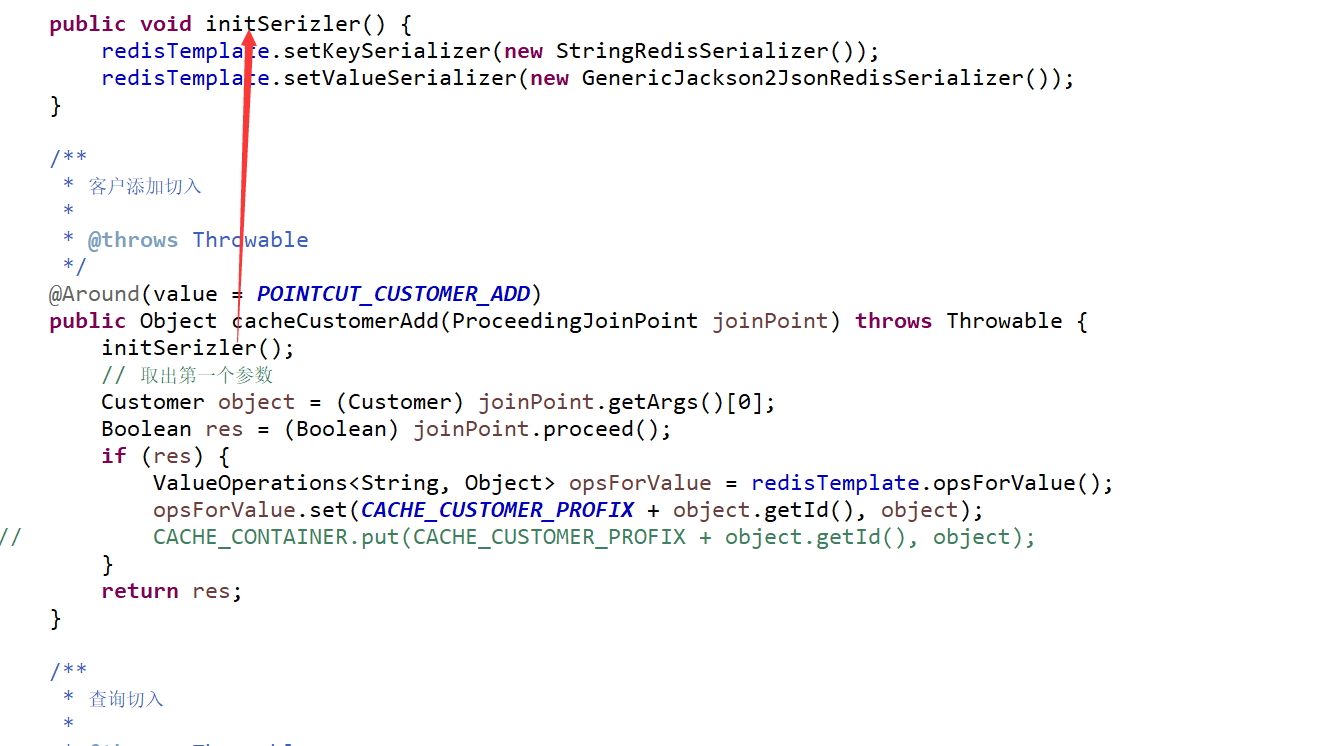


### 配置yml



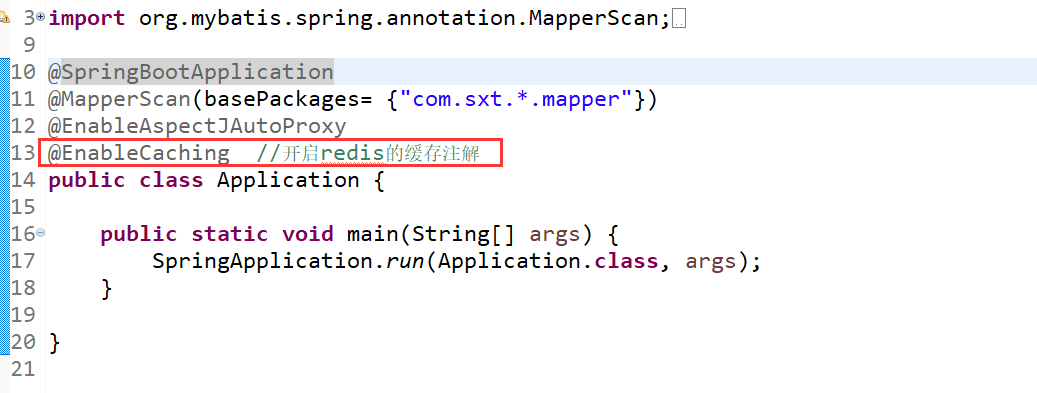
### 修改切面类



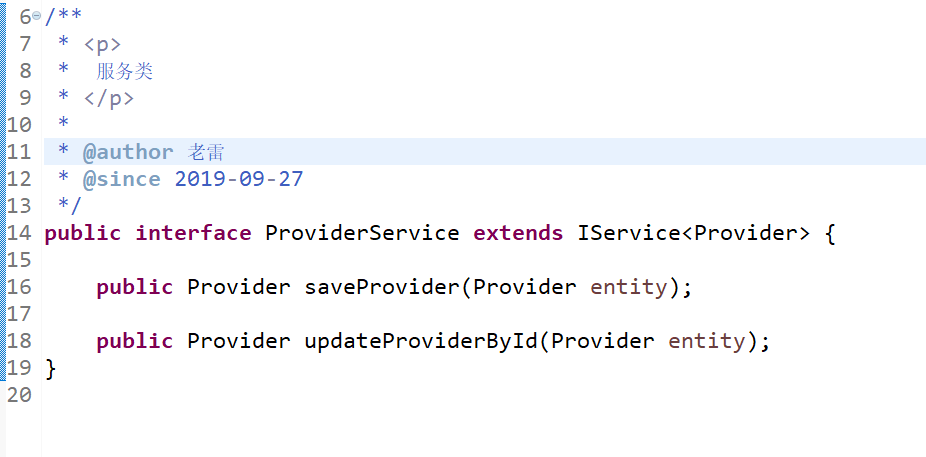


## 使用spring提供的注解方式来操作redis

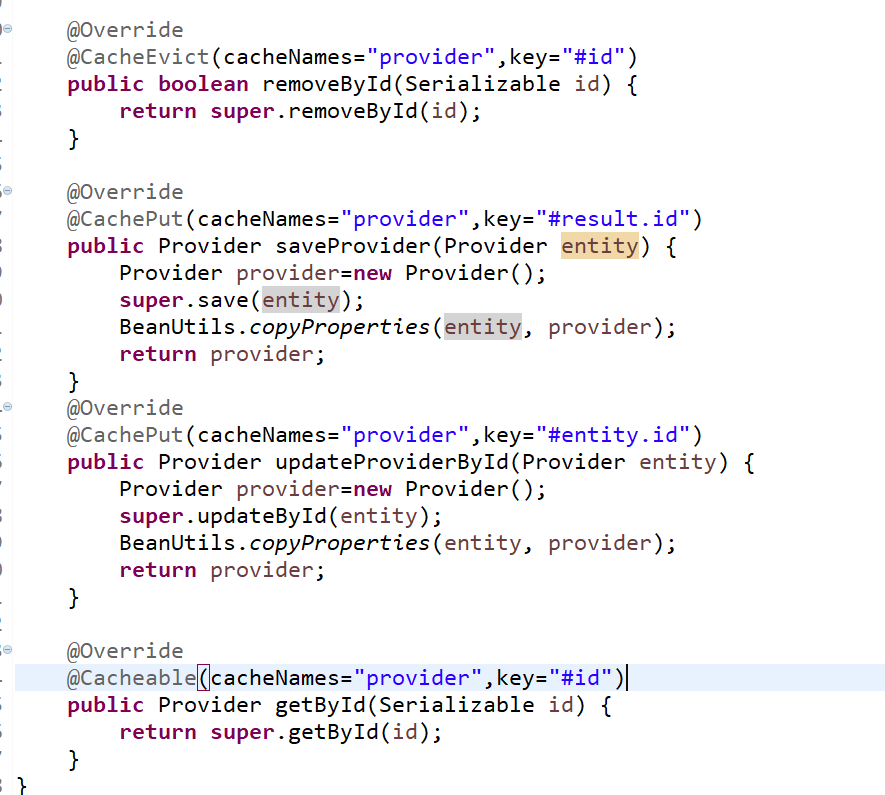
### 开启缓存注解



### 修改ProviderService



### 修改ProviderServiceImpl



### 修改ProviderContrller

### 更改序列化的方式[创建配置类去解决]

|  |
| --- |
| @Configuration  **public** **class** RedisConfig {  @Bean  **public** RedisCacheConfiguration redisCacheConfiguration(CacheProperties cacheProperties) {  CacheProperties.Redis redisProperties = cacheProperties.getRedis();  RedisCacheConfiguration config = RedisCacheConfiguration  .*defaultCacheConfig*();  config = config.serializeValuesWith(RedisSerializationContext.SerializationPair  //把默认的jdk的序列化方式变成jackson  .*fromSerializer*(**new** GenericJackson2JsonRedisSerializer()));  **if** (redisProperties.getTimeToLive() != **null**) {  config = config.entryTtl(redisProperties.getTimeToLive());  }  **if** (redisProperties.getKeyPrefix() != **null**) {  config = config.prefixKeysWith(redisProperties.getKeyPrefix());  }  **if** (!redisProperties.isCacheNullValues()) {  config = config.disableCachingNullValues();  }  **if** (!redisProperties.isUseKeyPrefix()) {  config = config.disableKeyPrefix();  }  **return** config;    }  } |

### 注意点

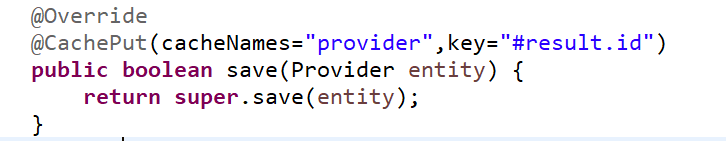
注解缓存缓存的是当前注解所在方法的返回值

cacheNames 缓存的前缀

key 代表:内容

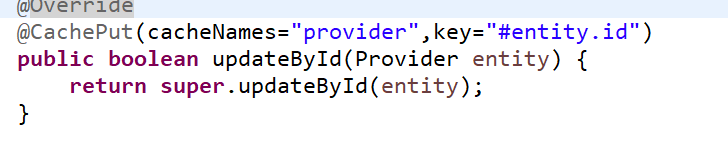
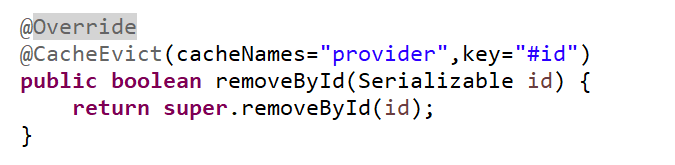
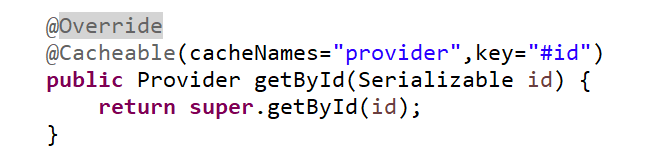
CachePut ===jedis.set方法

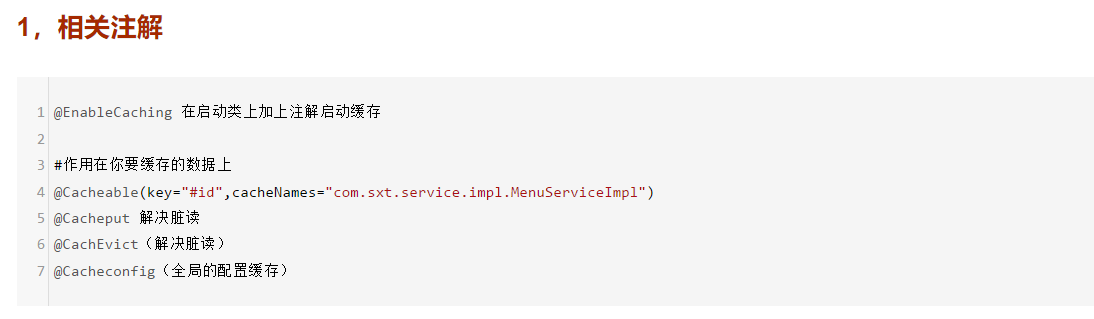
#result代表当前方法执行完成之后的返回值对象



entity.id

entity指当前方法的形式参数的名字 从里面可以取出id做为key



# 19【掌握】Redis键和值的设计原则

## 1、redis value 值格式

在Java常规开发中，我们需要有面向对象的思想，相对于对象来说，比较常用且能快速转换的格式就是 JSON 了；比较常用的Java处理JSON数据有三个比较流行的类库FastJSON、Gson和Jackson

上面提到了JSON，这是因为在Redis的存储中，我们使用它来存储value值，为什么要这样做呢？主要是因为json格式有如下几种好处：

标准，主流数据交换格式

简单，结构清晰，相对于XML来说更加的轻量级，易于解析

语言无关，任何语言都能轻松搞它

类型安全，值是有类型的，比如整数、字符串、布尔等

代码中redis value在存储前我们对其做了一次转换，将对象V转换为json对象后存储

也就是一个key对应一个json串

## 2、redis key 键格式

上面讲了简单的key存储，如 xdd的存储，此时普通的需求可以满足；然而在实际业务中，往往key键的存储会非常的复杂，比如我们现在有一个需求：

需求：根据基础数据系统中的数据字典类型查询对应的字典集合

这时，我们需要关注的业务就变得复杂了，就不能使用常规的key键存储方式，上面的需求大致可以拆分为：

系统：基础数据系统 user:sex:1=男 user:sex:0= 女

模块：数据字典 sys:available:1=true sys:available:0=false

方法：根据数据字典类型查询

参数：字典类型

为什么要这样拆分呢？为了可读性；也为了抽象出key存储规则；因为业务复杂情况下，我们定义的key键太多时就不便于管理，也不便于查找，以 系统-模块-方法-参数 这样的规则定义，我们可以很清晰的了解redis key存储的值是做了什么事情

common:sys:sex:1 男

common:sys:sex:0 女

common:sys:flag:1 是

common:sys:flag:0 否

common:page:title 欢迎使用XX管理系统

user:1 {id:1.name:小明}

user:2 {id:2.name:习大大}

这个在使用工具去查看的时候就可以看出层级关系啦

# 20【掌握】面试中要知道的

## 1，Redis支持的数据类型？

String（字符串类型）String数据结构是简单的key-value类型，value其实不仅可以是String，也可以是数字。 常规key-value缓存应用； 常规计数：微博数，粉丝数等。

常用命令: set,get,incr,decr,mget 等：set key value 设置值、 get key 获取值、 incr key 加一、 decr key 减一

​

hash（哈希）

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，后续操作的时候，你可以直接仅仅修改这个对象中的某个字段的值。

常用命令: set,get,decr,incr,mget 等：

hset key field value 设置值

hget key field 获取值

hincrby key field num 设置增数量

list（列表）

Redis list 的实现为一个双向链表，即可以支持反向查找和遍历，更方便操作，不过带来了部分额外的内存开销。

Redis list 的应用场景非常多，也是Redis最重要的数据结构之一，比如微博的关注列表，粉丝列表，消息列表等功能都可以用Redis的 list 结构来实现。

可以通过 lrange 命令，就是从某个元素开始读取多少个元素，可以基于 list 实现分页查询，这个很棒的一个功能，基于 redis 实现简单的高性能分页，可以做类似微博那种下拉不断分页的东西（一页一页的往下走），性能高。

常用命令: lpush,rpush,lpop,rpop,lrange等：

lpush list a b c d (从list左边添加元素)、 rpush list 1 2 3 4 (从list右边添加元素)

lrange list 0 -1(从0 到 -1 元素查看：也就表示查看所有)

lpop list （从list左边取，删除）、 rpop list (从list右边取，删除)

set（集合）

Redis的Set是string类型的无序集合。集合是通过哈希表实现的，所以添加，删除，查找的复杂度都是O(1)。

常用命令： sadd,spop,smembers,sunion 等：

sadd set1 a b c d d (向set1中添加元素) 元素不重复

smembers set1（查询元素）、 srem set1 a（删除元素）

sorted set（zset，有序集合）

和set相比，sorted set增加了一个权重参数score，使得集合中的元素能够按score进行有序排列。

例：在直播系统中，实时排行信息包含直播间在线用户列表，各种礼物排行榜，弹幕消息（可以理解为按消息维度的消息排行榜）等信息，适合使用 Redis 中的 Sorted Set 结构进行存储。

常用命令： zadd,zrange,zrem,zcard等：

zadd zset1 1 a 2 b 3 c （添加元素 zadd key score member，这里添加元素a:1分、元素b:2分、元素c:3分 ）

zrange zset1 0 -1 (查看zset1的所有元素，默认从小到大)

zrange zset1 0 -1 withscores (查看zset1的所有元素,包括分数score)

zrevrange zset1 0 -1 (查看zset1的所有元素，从大到小)

zincrby zset1 5 a (对zset1的a元素增加5分)

## 2，什么是Redis持久化？Redis有哪几种持久化方式？优缺点是什么

持久化就是把内存的数据写到磁盘中去，防止服务宕机了内存数据丢失。

　　（Redis 数据都放在内存中。如果机器挂掉，内存的数据就不存在。所以需要做持久化，将内存中的数据保存在磁盘，下一次启动的时候就可以恢复数据到内存中。）

　　Redis 提供了两种持久化方式:RDB（默认） 和AOF 。

RDB （快照）：

　　Redis可以通过创建快照来 获得存储在内存里面的数据在某个时间点上的副本。Redis创建快照之后，可以对快照进行备份，可以将快照复制到其他服务器从而创建具有相同数据的服务器副本（Redis主从结构，主要用来提高Redis性能），还可以将快照留在原地以便重启服务器的时候使用。

　　快照持久化是Redis默认采用的持久化方式，在redis.conf配置文件中默认有此下配置：

save 900 1 #在900秒(15分钟)之后，如果至少有1个key发生变化，Redis就会自动触发BGSAVE命令创建快照。

save 300 10 #在300秒(5分钟)之后，如果至少有10个key发生变化，Redis就会自动触发BGSAVE命令创建快照。

save 60 10000 #在60秒(1分钟)之后，如果至少有10000个key发生变化，Redis就会自动触发BGSAVE命令创建快照。

AOF（只追加文件）：

　　与快照持久化相比，AOF持久化的实时性更好，因此已成为主流的持久化方案。默认情况下Redis没有开启AOF（append only file）方式的持久化，可以通过appendonly参数开启：appendonly yes

　　开启AOF持久化后每执行一条会更改Redis中的数据的命令，Redis就会将该命令写入硬盘中的AOF文件。AOF文件的保存位置和RDB文件的位置相同，都是通过dir参数设置的，默认的文件名是appendonly.aof。

　　在Redis的配置文件中存在三种不同的 AOF 持久化方式，它们分别是：

appendfsync always #每次有数据修改发生时都会写入AOF文件,这样会严重降低Redis的速度

appendfsync everysec #每秒钟同步一次，显示地将多个写命令同步到硬盘

appendfsync no #让操作系统决定何时进行同步

　　为了兼顾数据和写入性能，用户可以考虑 appendfsync everysec选项 ，让Redis每秒同步一次AOF文件，Redis性能几乎没受到任何影响。而且这样即使出现系统崩溃，用户最多只会丢失一秒之内产生的数据。当硬盘忙于执行写入操作的时候，Redis还会优雅的放慢自己的速度以便适应硬盘的最大写入速度。

RDB （快照）：快照形式 ，定期将当前时刻的数据保存磁盘中。会产生一个dump.rdb文件

特点：性能较好，数据备份。但可能会存在数据丢失。

AOF（只追加文件） ：append only file (所有对redis的操作命令记录在aof文件中)，恢复数据，重新执行一遍即可。

特点：每秒保存，数据比较完整。但耗费性能。

　　【注】如果两个都配了优先加载AOF。（同时开启两个持久化方案，则按照 AOF的持久化放案恢复数据。）

## 3，Redis 有哪些架构模式？讲讲各自的特点？

主从模式（redis2.8版本之前的模式）、哨兵sentinel模式（redis2.8及之后的模式）、redis cluster模式（redis3.0版本之后）

## 4，什么是缓存穿透？如何避免？什么是缓存雪崩？何如避免？

缓存穿透

一般的缓存系统，都是按照key去缓存查询，如果不存在对应的value，就应该去后端系统查找（比如DB）。一些恶意的请求会故意查询不存在的key,请求量很大，就会对后端系统造成很大的压力。这就叫做缓存穿透。

如何避免？

1：对查询结果为空的情况也进行缓存，缓存时间设置短一点，或者该key对应的数据insert了之后清理缓存。

2：对一定不存在的key进行过滤。可以把所有的可能存在的key放到一个大的Bitmap中，查询时通过该bitmap过滤

3：也可以使用流行的bloom filter布隆过滤器

缓存雪崩

当缓存服务器重启或者大量缓存集中在某一个时间段失效，这样在失效的时候，会给后端系统带来很大压力。导致系统崩溃。

如何避免？

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

2：做二级缓存，A1为原始缓存，A2为拷贝缓存，A1失效时，可以访问A2，A1缓存失效时间设置为短期，A2设置为长期

3：不同的key，设置不同的过期时间，让缓存失效的时间点尽量均匀。