# Redis背景介绍

## 特点

使用C与语言开发的

单进程单线程的

官方提供的数据是可以达到100000+的qps

## Redis快的主要原因：

完全基于内存

数据结构简单，对数据操作也简单

使用多路 I/O 复用模型

多路 I/O 复用模型是利用select、poll、epoll可以同时监察多个流的 I/O 事件的能力，在空闲的时候，会把当前线程阻塞掉，当有一个或多个流有I/O事件时，就从阻塞态中唤醒，于是程序就会轮询一遍所有的流（epoll是只轮询那些真正发出了事件的流），并且只依次顺序的处理就绪的流，这种做法就避免了大量的无用操作。这里“多路”指的是多个网络连接，“复用”指的是复用同一个线程。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），且Redis在内存中操作数据的速度非常快（内存内的操作不会成为这里的性能瓶颈），主要以上两点造就了Redis具有很高的吞吐量。

# 与spring整合使用

## 配置

applicationContext.xml

### 1.连接池配置

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

<!-- 最大空闲连接数 -->

<property name=*"maxIdle"* value=*"100"*/>

<!-- 获取连接时的最大等待毫秒数(如果设置为阻塞时BlockWhenExhausted),如果超时就抛异常, 小于零:阻塞不确定的时间, 默认-1 -->

<property name=*"maxWaitMillis"* value=*"10000"*/>

<property name=*"testOnBorrow"* value=*"true"*/>

</bean>

JedisPoolConfig extends GenericObjectPoolConfig extends BaseObjectPoolConfig

BaseObjectPoolConfig集成的参数：

blockWhenExhausted 连接耗尽时是否阻塞，默认true

jmxEnabled 是否使用pool的jmx管理功能，默认true

jmxNamePrefix 默认pool

lifo 是否启用先进先出，默认true

maxWaitMillis 获取连接时最大等待毫秒数，默认-1

minEvictableIdleTimeMillis 逐出连接最小空闲时间，默认30分钟

softMinEvictableIdleTimeMillis 对象空闲多久后直接逐出，默认30分钟

testOnBorrow 在获取连接时检查有效性，默认false

testOnReturn

testWhileIdle 在空闲时检查有效性，默认false

timeBetweenEvictionRunsMillis 逐出扫描的间隔毫秒数，默认-1，不扫描

numTestsPerEvictionRun 每次逐出检查时，逐出的最大连接数，如果是-1，就是1

GenericObjectPoolConfig集成的参数

maxTotal 最大连接数，默认8个

maxIdle 最大空闲连接数，默认8个，超过这个值将进行回收

minIdle 最小空闲连接数，默认0个，低于这个值就创建新的

JedisPoolConfig干了这些事

setTestWhileIdle(true);

setMinEvictableIdleTimeMillis(60000);

setTimeBetweenEvictionRunsMillis(30000);

setNumTestsPerEvictionRun(-1);

# 连接工厂配置

<bean id=*"jedisConnectionFactory"* class=*"org.springframework.data.redis.connection.jedis.JedisConnectionFactory"*>

<property name=*"hostName"* value=*"${redis.hostName}"*/>

<property name=*"port"* value=*"${redis.port}"*/>

<!-- 上线这句要注释掉 -->

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

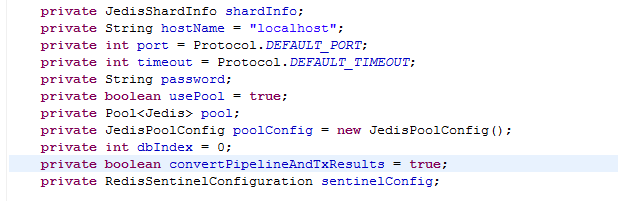
<property name=*"poolConfig"* ref=*"jedisPoolConfig"*/>

<property name=*"timeout"* value=*"1000"*/>

<property name=*"database"* value=*"${redis.database}"*/>

</bean>

JedisConnectionFactory



hostname 连接的主机地址

port 端口号

password 密码

timeout 超时时间，默认2000毫秒

pool jedis连接池

usePool 默认true

poolConfig 就是JedispoolConfig，上面配置的那个

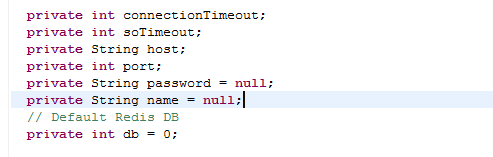
convertPipelineAndTxResults 默认true，暂不清楚此配置干嘛的

shardInfo 分片信息

sentinelConfig 暂不清楚此配置干嘛的

dbIndex 使用的是哪个库0-15

JedisShardInfo extends ShardInfo<Jedis>



这块的配置等做集群的时候补上

# 模板类redisTemplate、stringRedisTemplate

**属于spring与redis客户端整合的类，stringRedisTemplate继承自redisTemplate**

<bean id=*"redisTemplate"* class=*"org.springframework.data.redis.core.RedisTemplate"*

p:connectionFactory-ref=*"jedisConnectionFactory"*

p:keySerializer-ref=*"stringRedisSerializer"*

p:hashKeySerializer-ref=*"stringRedisSerializer"*

p:hashValueSerializer-ref=*"stringRedisSerializer"*

/>

<bean id=*"stringRedisTemplate"* class=*"org.springframework.data.redis.core.StringRedisTemplate"*

p:connectionFactory-ref=*"jedisConnectionFactory"*

/>

RedisTemplate<K, V> **extends** RedisAccessor **implements** RedisOperations<K, V>

## 父类RedisAccessor继承来的属性

connectionFactory连接工厂类

## 接口RedisOperations定义的方法

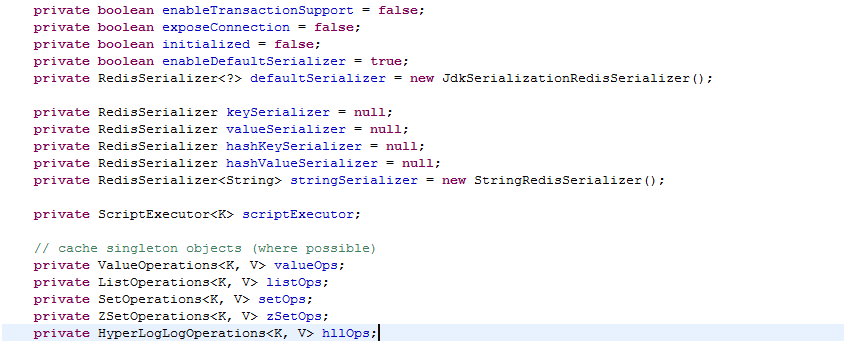
<T> T execute(RedisCallback<T> action);

<T> T execute(SessionCallback<T> session);

List<Object> executePipelined(RedisCallback<?> action);

…

## RedisTemplate自己的属性



enableTransactionSupport 是否支持事务处理，默认false

exposeConnection

initialized

enableDefaultSerializer

## 序列化类

<bean id=*"stringRedisSerializer"*

class=*"org.springframework.data.redis.serializer.StringRedisSerializer"*/>

# 不使用spring，直接连接redis

工具类

JedisPool extends Pool<Jedis>

JedisPoolConfig config=new JedisPoolConfig();

config.setMaxIdle(5);

config.setMaxWaitMillis(1000\*100);

config.setTestOnBorrow(true);

pool=new JedisPool(config,"192.168.10.22",6379);

Pool提供了getResource方法，返回一个redis连接，Jedis

Jedis **extends** BinaryJedis **implements** JedisCommands, MultiKeyCommands,

AdvancedJedisCommands, ScriptingCommands, BasicCommands, ClusterCommands, SentinelCommands

Jedis里面提供了各种redis客户端的命令方法入口，如get、set方法等等

# redis安装配置

## 安装

1.版本：redis-2.8.3.tar.gz

c语言的源代码包

2.安装之前确定已经安装gcc

[hadoop@master2 root]$ gcc

gcc: no input files

说明已经安装了

3.创建目录

mkdir /application/hadoop/cache

chmod 777 /application/hadoop/cache

4.上传redis-2.8.3.tar.gz到 /application/hadoop/cache下

chown -R hadoop:hadoop redis-2.8.3.tar.gz

5.解压缩

tar -zxvf redis-2.8.3.tar.gz

进入目录：cd redis-2.8.3

7.编译

make PREFIX=/application/hadoop/cache/redis install

8.启动

加配置文件启动：redis-server redis.conf &

需要将redis.conf拷贝到：/application/hadoop/cache/redis/bin

redis.conf目录/application/hadoop/cache/redis-2.8.3

## 配置文件说明

redis.conf

### 基本设置

daemonize：是否在后台运行的，默认no，如果需要在后台运行，把该项的值更改为yes

pidfile：pid文件存放路径，默认/var/run/redis\_6379.pid，当运行多个redis服务时需要指定不同的pid文件地址和端口

port:端口号

bind ：127.0.0.1，绑定127.0.0.1代表只能本机客户端访问，绑定0.0.0.0代表可以远程访问，配合密码一起使用来解决安全问题，一般配置密码+绑定0.0.0.0

**timeout：**默认是0设置客户端连接时的超时时间，单位为秒。当客户端在这段时间内没有发出任何指令，那么server端关闭该连接。0为关闭该设置。

tcp-keepalive：默认是**0，**指定TCP连接是否为长连接,”侦探”信号由server端维护,长连接将会额外的增加server端的开支,默认为0.表示禁用,非0值表示开启”长连接” ;”侦探”信号的发送间隔将有linux系统决定在多次”侦探”后,如果对等端仍不回复,将会关闭连接,否则连接将会被保持开启.client端socket也可以通过配置keepalive选项,开启”长连接”.

**loglevel** 默认**notice**  
server日志级别,合法值:debug,verbose,notice,warning 默认为notice  
debug适合开发环境,客户端操作信息都会输出日志  
verbose输出一些相对有用的信息,目前效果不明  
notice适合生产环境  
warning异常信息

**logfile**  
指定Redis日志记录方式，默认值为stdout，表示打印在命令行终端的窗口上，也可设为/dev/null屏蔽日志

databases **16**  
设定redis所允许的最大”db簇”（可以理解为数据库）的个数,默认为16个簇.  
客户端可以通过”select”指令指定需要使用的”db簇”索引号,默认为0.  
redis的顶层数据结构中,所有K-V都潜在的包括了”db簇”索引号,任何一个key都将隶属于一个”db”.  
任何对数据的检索,只会覆盖指定的”db”;例如数据被插入到”db 10″中,那么在”db 1″中去get,将会返回null.  
对数据归类到不同的db簇中,可以帮助我们实现一些特定的需求,比如根据不同客户端连接,来指定不同的db索引号.

maxclients 128  
限制同时连接的客户数量。  
当连接数超过这个值时，redis 将不再接收其他连接请求，客户端尝试连接时将收到 error 信息

设置为2时候的会显示一下错误  
Error: Connection reset by peer

**maxmemory <bytes>**  
设置redis能够使用的最大内存。  
达到最大内存设置后，Redis会先尝试清除已到期或即将到期的Key（设置过expire信息的key）  
在删除时,按照过期时间进行删除，最早将要被过期的key将最先被删除  
如果已到期或即将到期的key删光，仍进行set操作，那么将返回错误  
此时redis将不再接收写请求,只接收get请求。  
maxmemory的设置比较适合于把redis当作于类似memcached 的缓存来使用

maxmemory-policy  
当内存达到最大值的时候Redis会选择删除哪些数据？有六种方式可供选择  
volatile-lru -> 利用LRU算法移除设置过过期时间的key (LRU:最近最少使用 Least Recently Used )  
allkeys-lru -> 利用LRU算法移除任何key  
volatile-random -> 移除设置过过期时间的随机key  
allkeys->random -> remove a random key, any key  
volatile-ttl -> 移除即将过期的key(minor TTL)  
noeviction -> 不移除任何可以，只是返回一个写错误  
注意：对于上面的策略，如果没有合适的key可以移除，当写的时候Redis会返回一个错误  
写命令包括: set setnx setex append  
incr decr rpush lpush rpushx lpushx linsert lset rpoplpush sadd  
sinter sinterstore sunion sunionstore sdiff sdiffstore zadd zincrby  
zunionstore zinterstore hset hsetnx hmset hincrby incrby decrby  
getset mset msetnx exec sort  
  
默认是:  
# maxmemory-policy volatile-lru

**syslog-enabled**   
'syslog-enabled'设置为yes会把日志输出到系统日志，默认是no

syslog-ident redis  
指定syslog的标示符，如果'syslog-enabled'是no，则这个选项无效。

### slowLog

**1. slowlog-log-slower-than** 默认10000，此为微秒单位，即1毫秒  
Redis slow log用来记录超过指定执行时间的查询。执行时间不包括I/O计算比如连接客户端，返回结果等，只是命令执行时间可以通过两个参数设置slow log：一个是告诉Redis执行超过多少时间被记录的参数slowlog-log-slower-than(毫秒)，另一个是slow log的长度。当一个新命令被记录的时候最早的命令将被从队列中移除，负数则关闭slow log，0则会导致每个命令都被记录

**2. slowlog-max-len 默认128个**  
对日志长度限制

#### Redis Slowlog读取

redis-cli客户端通过slowlog get指令获取最新10条慢查询命令。

当然各语言的client也实现对应的接口。

示例：获取最近2个慢查询命令

127.0.0.1:6381> SLOWLOG **get** 2

1) 1) (integer) 6

2) (integer) 1458734263

3) (integer) 74372

4) 1) "hgetall"

2) "max.dsp.blacklist"

2) 1) (integer) 5

2) (integer) 1458734258

3) (integer) 5411075

4) 1) "keys"

2) "max.dsp.blacklist"

分析slowlog query：

以第一个HGET命令为例分析，每个slowlog实体共4个字段：

\* 字段1：1个整数,表示这个slowlog出现的序号,server启动后递增, 当前为6.

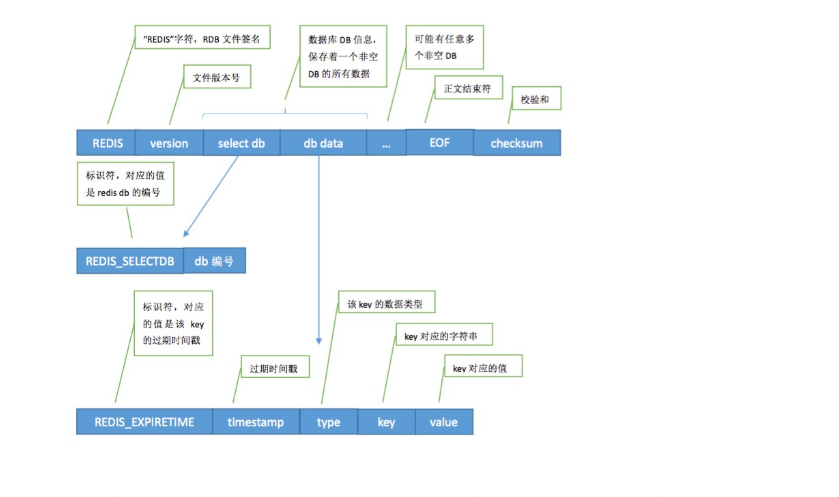
\* 字段2：表示查询执行时的Unix时间戳.

\* 字段3：表示查询执行微妙数,当前是74372微妙,约74ms.

\* 字段4: 表示查询的命令和参数,如果参数很多或很大,只会显示部分并给数参数个数;

当前命令是"hgetall" "max.dsp.blacklist"

### Snapshotting



**save,**默认

save 900 1 15分钟内，有一个变更操作，触发snapshot

save 300 10 5分钟内，有10个变更操作

save 60 10000 1分钟内，有10000个变更操作

rdbcompression **yes**  
是否启用rdb文件压缩手段,默认为yes.  
压缩可能需要额外的cpu开支,不过这能够有效的减小rdb文件的大小,有利于存储/备份/传输/数据恢复.

**rdbchecksum yes**  
是否对rdb文件使用CRC64校验和,默认为”yes”,那么每个rdb文件内容的末尾都会追加CRC校验和.对于其他第三方校验工具,可以很方便的检测文件的完整性

**dbfilename** dump.rdb镜像备份文件的文件名

dir **./** 指定rdb/AOF文件的目录位置，只能为文件夹不能为文件

实现原理：

Redis提供了save和bgsave两个命令来生成RDB文件

Save：由主进程来完成，会阻塞redis

Bgsave:会先fork一个子进程，在子进程中生成RDB文件

### AOF

**1.** appendonly no  
#默认情况下，Redis会异步的把数据保存到硬盘。如果你的应用场景允许因为系统崩溃等极端情况而导致最新数据丢失的话，那这种做法已经很ok了。否则你应该打开‘append only’模式，开启这种模式后，Redis会在#appendonly.aof文件中添加每一个写操作，这个文件会在Redis启动时被读取来在内存中重新构建数据集。  
注意：如果你需要，你可以同时开启‘append only’模式和异步dumps模式（你需要注释掉上面的‘save’表达式来禁止dumps），这种情况下，Redis重建数据集时会优先使用appendonly.aof而忽略dump.rdb

**2.** appendfilenameappendonly.aof  
AOF文件名称 (默认: "appendonly.aof")

**3.** appendfsynceverysec（文件同步）  
# 调用fsync()函数通知操作系统立刻向硬盘写数据,就是将redis的操作命令写到aof文件中去  
# Redis支持三种同步AOF文件的策略:  
#  
# no:不fsync, 只是通知OS可以flush数据了，具体是否flush取决于OS.性能更好.  
# always: 每次写入append only 日志文件后都会fsync . 性能差，但很安全.  
# everysec: 没间隔1秒进行一次fsync. 折中.  
#  
# 默认是"everysec"，按照速度和安全折中这是最好的。  
# 如果想让Redis能更高效的运行，你也可以设置为"no"，让操作系统决定什么时候去执行  
# 或者相反想让数据更安全你也可以设置为"always"  
#  
# 如果不确定就用 "everysec".

**4.** no-appendfsync-on-rewrite no（文件重写）  
# AOF策略设置为always或者everysec时，后台处理进程(后台保存或者AOF日志重写)会执行大量的I/O操作  
# 在某些Linux配置中会阻止过长的fsync()请求。注意现在没有任何修复，即使fsync在另外一个线程进行处理  
#  
# 为了减缓这个问题，可以设置下面这个参数no-appendfsync-on-rewrite

**5.** auto-aof-rewrite-percentage100  
auto-aof-rewrite-min-size 64mb  
# append only 文件的自动重写  
# 当AOF 日志文件即将增长到指定百分比时，Redis可以通过调用BGREWRITEAOF 来自动重写append only文件。  
# 它是这么干的：Redis会记住最近一次重写后的AOF 文件size。然后它会把这个size与当前size进行比较，如果当前size  
比指定的百分比大，就会触发重写。同样，你需要指定AOF文件被重写的最小size，这对避免虽然百分比达到了, 但是实  
际上文件size还是很小（这种情况没有必要重写）却导致AOF文件重写的情况很有用。  
# auto-aof-rewrite-percentage 设置为 0 可以关闭AOF重写功能

#AOF重写并不需要对原有AOF文件进行任何的读取，写入，分析等操作，这个功能是通##过读取服务器当前的数据库状态来实现的

原理：

#通过子进程的方式重写

AOF缓冲区：redis写命令准备写入磁盘的缓冲区

AOF重写缓冲区：aof文件重写过程中，redis写命令缓冲的内存区域，待AOF重写完，这部分命令会追加到重写后的aof文件中。

子进程重写过程中，主进程执行下面操作：

执行client发来的命令请求；

将写命令追加到现有的AOF文件中；

将写命令追加到AOF重写缓存中。

### 高级配置

list-max-ziplist-entries 512 **list-max-ziplist-value 64**  
list 数据类型多少节点以下会采用去指针的紧凑存储格式。  
list 数据类型节点值大小小于多少字节会采用紧凑存储格式。

hash-max-ziplist-entries 512

hash-max-ziplist-value 64

**activerehashing yes**  
是否重置Hash表  
设置成yes后redis将每100毫秒使用1毫秒CPU时间来对redis的hash表重新hash，可降低内存的使用  
当使用场景有较为严格的实时性需求,不能接受Redis时不时的对请求有2毫秒的延迟的话，把这项配置为no。  
如果没有这么严格的实时性要求,可以设置为 yes,以便能够尽可能快的释放内存

set-max-intset-entries 512  
set数据类型内部数据如果全部是数值型，且包含多少节点以下会采用紧凑格式存储。

zset-max-ziplist-entries 128 **zset-max-ziplist-value 64**  
zsort 数据类型多少节点以下会采用去指针的紧凑存储格式。  
zsort 数据类型节点值大小小于多少字节会采用紧凑存储格式。

### Replication（复制）

slaveof  
将当前server做为slave,并为其指定master信息.  
slaveof <masterip> <masterport> 当本机为从服务时，设置主服务的IP及端口

slaveof 11.41.8.209 63789

**masterauth**  
以认证的方式连接到master。 如果master中使用了”密码保护”，slave必须交付正确的授权密码，才能连接成功。  
“requirepass”配置项指定了当前server的密码。  
此配置项中值需要和master机器的”requirepass”保持一致

slave-serve-stale-data yes  
如果当前server是slave,那么当slave与master失去通讯时,是否继续为客户端提供服务,”yes”表示继续,”no”表示终止.  
在”yes”情况下,slave继续向客户端提供只读服务,有可能此时的数据已经过期.  
在”no”情况下,任何向此server发送的数据请求服务(包括客户端和此server的slave)都将被告知”error”，但 INFO 和SLAVEOF  
命令除外

### VM(虚拟内存)

1. vm-enabled no

是否开启虚拟内存支持。  
redis 是一个内存数据库，当内存满时,无法接收新的写请求,所以在redis2.0后,提供了虚拟内存的支持  
但需要注意的，redis 所有的key都会放在内存中，在内存不够时,只把value 值放入交换区  
虽使用虚拟内存，但性能基本不受影响，需要注意的是要把vm-max-memory设置到足够来放下所有的key

2. vm-swap-file /tmp/redis.swap  
设置虚拟内存的交换文件路径，不可多个Redis实例共享

3. vm-max-memory 0  
设置开启虚拟内存后,redis将使用的最大物理内存大小。  
默认为0，redis将把他所有能放到交换文件的都放到交换文件中，以尽量少的使用物理内存  
即当vm-max-memory设置为0的时候,其实是所有value都存在于磁盘  
在生产环境下,需要根据实际情况设置该值,最好不要使用默认的 0

4. vm-page-size 32  
设置虚拟内存的页大小  
如果 value 值比较大,如要在 value 中放置博客、新闻之类的所有文章内容，就设大一点

5. vm-pages 134217728  
设置交换文件的总的 page 数量  
注意page table信息是放在物理内存中，每8个page 就会占据RAM中的 1 个 byte  
总的虚拟内存大小 = vm-page-size \* vm-pages

6. vm-max-threads 4  
设置 VM IO 同时使用的线程数量。

### Redis官方使用VM建议

一般情况下**不建议**使用虚拟内存，如果有特殊需求，请考虑以下几种情况：

* 当key很小而value很大时,使用VM的效果会比较好.因为这样节约的内存比较大
* 当key不小时,可以考虑使用一些非常方法将很大的key变成很大的value,如可将key,value组合成一个新的value
* 最好使用linux ext3 等对稀疏文件支持比较好的文件系统保存你的swap文件
* vm-max-threads参数可设置访问swap文件的线程数，最好不要超过机器的核数；设置为0则所有对swap文件的操
* 作都是串行的，可能会造成比较长时间的延迟,但是对数据完整性有很好的保证

**redis数据存储**  
redis的存储分为内存存储、磁盘存储和log文件三部分，配置文件中有三个参数对其进行配置。

* save seconds updates，save配置，指出在多长时间内，有多少次更新操作，就将数据同步到数据文件。可多个条件配合，默认配置了三个条件。
* appendonly yes/no ，appendonly配置，指出是否在每次更新操作后进行日志记录，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的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。
* appendfsync no/always/everysec ，appendfsync配置，no表示等操作系统进行数据缓存同步到磁盘，always表示每次更新操作后手动调用fsync()将数据写到磁盘，everysec表示每秒同步一次。

### Security（安全）

**requirepass** Ygbx1234  
当前server的授权密码  
任何客户端或者slave与此server交互前,需要提交密码,其他server的masterauth配置和此参数值保持一致  
密码应该足够复杂(64字节)

如登录客户端使用命令./redis-cli –a Ygbx1234

rename-command  
命令重命名.  
例如: rename-command CONFIG b840fc02d524045429941cc15f59e41cb7be6c52

### Redis官方VM使用建议

### 登录redis

1.可以在redis的bin路径下用./redis-cli连接

2.如果redis.conf配置文件里bind是本机ip地址，不是127.0.0.1，可以用./redis-cli –h 192.168.1.22连接

3.如果配置文件配置了密码，那么可以用./redis-cli –a Ygbx1234连接

链接测试154redis

命令：redis-cli -a Ygbx1234

redis-cli -h 10.46.66.13 -p 6379 -a Ygbx1234

# redis数据结构介绍

## 字符串String

对应的命令有:

get

set：如果 key 已经存储其他值， SET 就覆写旧值，且无视类型。

del

incr 示例：将键存储的值加1incr key

decr示例：将键存储的值减1 decr key

incrby示例：将键存储的值加上整数amount incrby key amount

decrby示例：将键存储的值减去整数amount decrby key amount

incrbyfloat示例：将键存储的值加上浮点数amount,这个命令在redis2.6版本以上才可以用

append示例：将值value追加到给定key-name当前存储的值末尾，append key-name value

getrange示例：获取一个由偏移量start至偏移量end范围内所有字符组成的子串，包含start和end，getrange key-name start end

setrange 用法同上命令，设置指定偏移量的值

mset：MSET是原子的，所以所有给定的keys是一次性set的。客户端不可能看到这种一部分keys被更新而另外的没有改变的情况。MSET key1 "Hello" key2 "World"

## LIST列表

相当于java中的ArrayList集合

我的理解就是：一个key对应很多个value，并且value的值还可以重复，都有各自的位置

对应的命令有：

lpush:向列表左边（就是最前面的意思）增加一个元素，示例：lpush +key +value

rpush:向列表右边（就是最后面的意思）增加一个元素，示例：rpush +key+value

lpop:从列表的左边（就是前面）移除并返回一个元素，示例lpop +key+value

rpop:从列表的右边（就是后面）移除并返回尾元素。，示例rpop +key+value

lindex:获取列表在指定位置上的一个元素，示例：lindex+key+index

lrange:获取列表在指定范围内的所有元素，示例：lrange key 0 -1

0 -1 代表查所有的元素，也可以查0 2 这3个元素

ltrim :ltrim key-name start end 对列表进行修剪，保留start到end偏移量的元素，包含start和end元素

blpop示例：blpop key-name timeout 从列表中弹出一个元素，如果列表为空则阻塞timeout时间等待可弹出的元素出现

brpop：用法与上面命令一样，一个是从左边开始，一个是从右边开始

rpoplpush示例：rpoplpush source-key dest-key，从source-key列表的最右端弹出一个元素，然后将这个元素推入dest-key的最左端，并向用户返回这个元素

brpoplpush示例：rpoplpush source-key dest-key timeout，如果source-key为空，那么阻塞timeout秒等待可弹出的元素出现

### 内部编码

　　列表类型的内部编码有两种:　　**编码名 编码描述**

　　ziplist(压缩列表)　　当列表的元素个数大于list-max-ziplist-entries配置(默认为512个),同时列表中每个元素的长度小于list-max-ziplist-value配置(默认为64字节).

　　linkedlist(链表)　　当列表的长度或值得大小不满足ziplist的要求,redis会采用linkedlist为列表的内部实现编码.

列表的使用场景有很多如: lpush+lpop=Stack(栈)、lpush+rpop=queue(队列)、lpush+brpop=message queue、lpush+ltrim=Capped Collection(有限集合)

## SET集合

相当于java中的Set集合

Redis的集合和列表都可以存储多个字符串，他们的不同在于，列表可以存储多个相同的字符串，而集合则通过散列表来保证自己存储的每个字符串都是各不相同的（这些散列表只有键，但没有与键相关联的值）

对应的命令有：

sadd :向集合中添加一个元素，示例：sadd+key+value，也可以同时添加多个value，sadd+key+value1 value2

srem:从集合中删除一个元素，示例：srem+key+value，也可以同时删除多个

sismember:检查一个元素是否存在于集合中，示例：smember+key+value

smembers:获取集合中所有的元素，示例：smembers+key

scard：返回集合包含元素的数量，若set数量为0或者key不存在，返回0

spop：随机的移除集合中的一个元素，并返回被移除的元素

smove示例：smove source-key dest-key item，如果集合source-key中包含item，那么从source-key中删除item并将item添加到dest-key中，成功返回1失败返回0

**用于组合和处理多个集合的redis命令**

sdiff示例：sdiff key-name [key-name…]，返回那么存在于第一个集合，但不存在于其他集合的元素，数学上的差集

sdiffstore示例：sdiffstore dest-key key-name [key-name…]，将差集存储到dest-key集合里

sinter交集

sinterstore将交集存储到dest-key中

sunit并集，即所有的元素

sunitstore存储

### 内部编码

　　集合类型的内部编码有两种:

　　编码名　　　　　　　　　　　　编码描述

　　intset(整数集合)　　　　当集合中的元素全是整数,且长度不超过set-max-intset-entries(默认为512个)时,redis会选用intset作为内部编码.

　　hashtable(哈希表)　　　当集合无法满足intset的条件时,redis会使用hashtable作为内部编码.

## hash散列

相当于map集合，键是各不相同，无序排列

Redis的散列可以存储多个键值对之间的映射。

对应的命令有：

hset:向散列中添加一个键值对，示例：hset hash-key sub-key value

如果字段是哈希表中的一个新建字段，并且值设置成功，返回 1 。 如果哈希表中域字段已经存在且旧值已被新值覆盖，返回 0

hmset:向散列中添加一个或多个键值对，示例：hset hash-key sub-key value

sub-key1 value1 …

hget:从散列中获取一个键值对，示例：hget hash-key sub-key

hmget：从散列中获取一个或多个键值对，示例：hmget hash-key sub-key sub-key2

hgetall:获取散列中所有的键值对，示例：hget hash-key

hdel:从散列中删除一个键值对，示例：hdel hash-key sub-key

hlen:返回散列包含的键值对的数量

hexists示例：hexists key-name key检查给定键key是否存在于散列中key-name中

hkeys示例：hkeys key-name，获取散列包含的所有键

hvalues示例：hvalues key-name，获取散列包含的所有值

hincrby示例：hincrby key-name key amount 将键key存储的值加上整数amount

hincrbyfloat

## 有序集合

和散列一样，都用于存储键值对，有序集合的键被称为成员（member），每个成员都是各不相同的，而有序集合的值则被称为分值（score），分值必须为浮点数。有序集合是redis里唯一既可以根据成员访问元素，又可以根据分值访问元素以及分值的排序来访问元素的结构。

对应的命令有：

zadd:将一个带有给定分值的元素添加到有序集合里，示例：zadd zset-key 728 member1，也可以同时添加多个zadd zset-key 728 member1 729 member2…

zrem：如果给定成员存在于有序集合中，那么删除这个成员，示例：zrem zset-key member1，也可以同时删除多个zrem zset-key member1 member2…

zcard:返回有序集合包含的成员数量

zincrby示例：zincrby key-name increment member 将member成员的分值加上increment

zrange：根据元素在有序集合中的位置，从有序集合里面获取多个元素，示例：zrange zset-key 0 -1 withscores

zrangebyscore：获取有序集合在给定分值范围内的所有元素，示例：zrangebyscore zset-key 0 800 withscores

zcount 示例：zcount key-name min max返回分值介于min和max之间的成员数量

**有序集合的范围型数据获取命令和范围型数据删除命令，以及并集命令和交集命令**

Zrev\* 逆序的意思

zrevrank:示例zrevrank key-name member返回有序集合成员member的排名，成员按照分值从大到小排列

zrevrange:示例zrevrange key-name start stop [withscores]返回有序集合给定排名范围内的成员，成员按照分值从大到小排列

zremrangebyrank 示例：zremrangebyrank key-name start stop 移除有序集合中排名介于start和stop之间的所有成员

zremrangebyscore移除有序集合中分值介于start和stop之间的所有成员

zinterstore示例：zinterstore dest-key key-count key [key…]

**用于处理过期时间的redis命令**

|  |  |
| --- | --- |
| persist | persist key-name 移除键的过期时间 |
| ttl | ttl key-name 查看给定键距离过期还有多少秒 -1代表没有过期时间 |
| exprise | exprise key-name seconds 给键设置过期时间 |
| expireat | Expireat key-name timestamp 将给定键的过期时间设置为给定的unix时间戳 |
| pttl | Pttl key-name 查看给定键距离过期时间还有多少毫秒，2.6版本以上使用  当 key 不存在时，返回 -2 。  当 key 存在但没有设置剩余生存时间时，返回 -1 |
| pexpire | Pexpire key-name milliseseconds 让给定键在指定毫秒数之后过期 |
| pexpireat | 将一个毫秒级精度的unix时间戳设置为给定键的过期时间 |

### 内部编码

　　编码名称　　　　　　　　编码描述

　　ziplist(压缩列表)　　　　当有序集合的元素小于zset-max-ziplist-entries配置(默认是128个),同时每个元素的值都小于zset-max-ziplist-value(默认是64字节)时,Redis会用ziplist来作为有序集合的内部编码实现,ziplist可以有效的减少内存的使用

　　skiplist(跳跃表)　　　　  当ziplist的条件不满足时,有序集合将使用skiplist作为内部编码的实现,来解决此时ziplist造成的读写效率下降的问题.

# Redis主从复制

修改Replication配置

slaveof <masterip> <masterport>

masterauth <master-password>

将对应的主redis的ip、端口、密码配置上即可

# redis主从切换

## sentinel

使用redis官方推荐的集群管理工具sentinel，使用一个或多个sentinel和Redis的master/slave可以组成一个群集，可以检测master实例是否存活，并在master实例发生故障时，将slave提升为master，并在老的master重新加入到sentinel的群集之后，会被重新配置，作为新master的slave。这意味着基于Redis sentinel的HA群集是能够自我管理的。

## sentine.conf详解

bind :192.168.1.22

port：当前sentinel运行的端口号、

dir：sentinel运行时的临时文件夹

sentinel monitor master001 192.168 . 110.101 6379 2 : Sentinel去监视一个名为 master001 的主redis实例，这个主实例的IP地址为本机地址 192.168 . 110.101 ，端口号为 6379 ，而将这个主实例判断为失效至少需要 2 个 Sentinel进程的同意，只要同意Sentinel的数量不达标，自动failover就不会执行

sentinel auth-pass master redis：配置redis的密码

sentinel down - after - milliseconds master001 30000 : 指定了Sentinel认为Redis实例已经失效所需的毫秒数。 当实例超过该时间没有返回PING，或者直接返回错误，那么Sentinel将这个实例标记为主观下线。只有一个 Sentinel进程将实例标记为主观下线并不一定会引起实例的自动故障迁移：只有在足够数量的Sentinel都将一个实例标记为主观下线之后，实例才会被标记为客观下线，这时自动故障迁移才会执行

sentinel parallel - syncs master001 1 ： 指定了在执行故障转移时，最多可以有多少个从Redis实例在同步新的主实例，在从Redis实例较多的情况下这个数字越小，同步的时间越长，完成故障转移所需的时间就越长

sentinel failover - timeout master001 180000 ： 如果在该时间（ms）内未能完成failover操作，则认为该failover失败

sentinel notification-script <master-name> <script-path> ： 指定sentinel检测到该监控的redis实例指向的实例异常时，调用的报警脚本。该配置项可选，但是很常用

**加在文件最后，方便后台运行和日志查看**

daemonize yes

logfile "/application/hadoop/cache/sentinel/26379.log"

sentinel服务启动：

在redis的bin路径下

./redis-sentinel /path/to/sentinel.conf &

## spring配置文件

<bean id=*"jedisConnectionFactory"* class=*"org.springframework.data.redis.connection.jedis.JedisConnectionFactory"*>

<property name=*"hostName"* value=*"192.168.1.22"*/>

<property name=*"password"* value=*"redis"*></property>

<property name=*"poolConfig"* ref=*"jedisPoolConfig"*/>

<property name=*"timeout"*  value=*"1000"*/>

<property name=*"database"* value=*"0"*/>

<constructor-arg ref=*"redisSentinelConfiguration"*/>

</bean>

<!-- 哨兵集群模式配置 -->

<bean id=*"redisSentinelConfiguration"* class=*"org.springframework.data.redis.connection.RedisSentinelConfiguration"*>

<property name=*"master"*>

<bean class=*"org.springframework.data.redis.connection.RedisNode"*>

<property name=*"name"* value=*"master"*></property>

</bean>

</property>

<property name=*"sentinels"*>

<set>

<bean class=*"org.springframework.data.redis.connection.RedisNode"*>

<constructor-arg name=*"host"* value=*"192.168.1.22"*></constructor-arg>

<constructor-arg name=*"port"* value=*"26379"*></constructor-arg>

</bean>

<bean class=*"org.springframework.data.redis.connection.RedisNode"*>

<constructor-arg name=*"host"* value=*"192.168.1.22"*/>

<constructor-arg name=*"port"* value=*"26380"*/>

</bean>

<bean class=*"org.springframework.data.redis.connection.RedisNode"*>

<constructor-arg name=*"host"* value=*"192.168.1.22"*/>

<constructor-arg name=*"port"* value=*"26381"*/>

</bean>

</set>

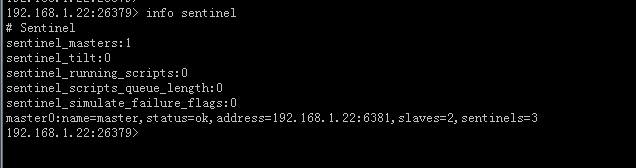
</property>

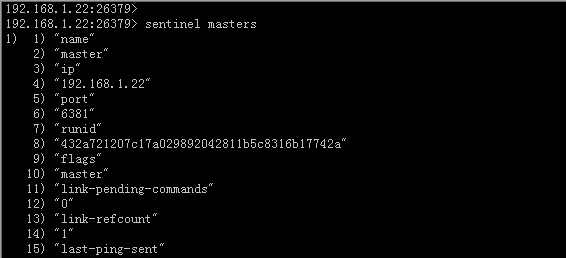
</bean>

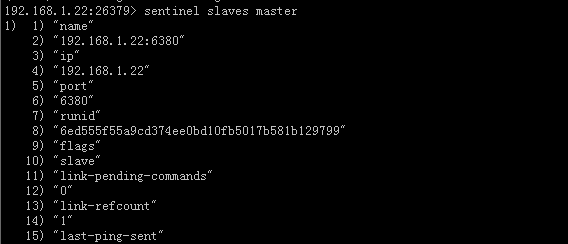
## 登录sentinel客户端、命令

./redis-cli –h 192.168.1.22 –p 26379

查看sentinel信息命令：







[PING](http://redis.readthedocs.org/en/latest/connection/ping.html#ping) ：返回 PONG 。

SENTINEL masters ：列出所有被监视的主服务器，以及这些主服务器的当前状态。

SENTINEL slaves <master name> ：列出给定主服务器的所有从服务器，以及这些从服务器的当前状态。

SENTINEL get-master-addr-by-name <master name> ： 返回给定名字的主服务器的 IP 地址和端口号。 如果这个主服务器正在执行故障转移操作， 或者针对这个主服务器的故障转移操作已经完成， 那么这个命令返回新的主服务器的 IP 地址和端口号。

SENTINEL reset <pattern> ： 重置所有名字和给定模式 pattern 相匹配的主服务器。 pattern 参数是一个 Glob 风格的模式。 重置操作清除主服务器目前的所有状态， 包括正在执行中的故障转移， 并移除目前已经发现和关联的， 主服务器的所有从服务器和 Sentinel 。

SENTINEL failover <master name> ： 当主服务器失效时， 在不询问其他 Sentinel 意见的情况下， 强制开始一次自动故障迁移 （不过发起故障转移的 Sentinel 会向其他 Sentinel 发送一个新的配置，其他 Sentinel 会根据这个配置进行相应的更新）。

# redis分布式集群cluster

## 集群介绍

### 集群目标

    1）高性能和线性扩展，最大可以支撑到1000个节点；Cluster架构中无Proxy层，Master与slave之间使用异步replication，且不存在操作的merge。（即操作不能跨多个nodes，不存在merge层）

    2）一定程度上保证writes的安全性，需要客户端容忍一定程度的数据丢失：集群将会尽可能（best-effort）保存客户端write操作的数据；通常在failover期间，会有短暂时间内的数据丢失（因为异步replication引起）；当客户端与少数派的节点处于网络分区时（network partition），丢失数据的可能性会更高。（因为节点有效性检测、failover需要更长的时间）

    3）可用性：只要集群中大多数master可达、且失效的master至少有一个slave可达时，集群都可以继续提供服务；同时“replicas migration”可以将那些拥有多个slaves的master的某个slave，迁移到没有slave的master下，即将slaves的分布在整个集群相对平衡，尽力确保每个master都有一定数量的slave备份。

    （Redis Cluster集群有多个shard(分片)组成，每个shard可以有一个master和多个slaves构成，数据根据hash slots配额分布在多个shard节点上，节点之间建立双向TCP链接用于有效性检测、Failover等，Client直接与shard节点进行通讯；Cluster集群没有Proxy层，也没有中央式的Master用于协调集群状态或者state存储；集群暂不提供动态reblance策略）

Nodes之间的通讯协议：gossip

### Cluster主要组件

keys分布模型

    集群将key分成16384个slots（hash 槽），slot是数据映射的单位，言外之意，Redis Cluster最多支持16384个nodes（每个nodes持有一个slot）。集群中的每个master持有16384个slots中的一部分，处于“stable”状态时，集群中没有任何slots在节点间迁移，即任意一个hash slot只会被单个node所服务（master，当然可以有多个slave用于replicas，slave也可以用来扩展read请求）。keys与slot的映射关系，是按照如下算法计算的：HASH\_SLOT = CRC16(key) mod 16384。其中CRC16是一种冗余码校验和，可以将字符串转换成16位的数字。

    hash tags

在计算hash slots时有一个意外的情况，用于支持“hash tags”；hash tags用于确保多个keys能够被分配在同一个hash slot中，用于支持multi-key操作。hash tags的实现比较简单，key中“{}”之间的字符串就是当前key的hash tags，如果存在多个“{}”，首个符合规则的字符串作为hash tags，如果“{}”存在多级嵌套，那么最内层首个完整的字符串作为hash tags，比如“{foo}.student”，那么“foo”是hash tags。如果key中存在合法的hash tags，那么在计算hash slots时，将使用hash tags，而不再使用原始的key。即“foo”与“{foo}.student”将得到相同的slot值，不过“{foo}.student”仍作为key来保存数据，即redis中数据的key仍为“{foo}.student”。

### 容错

## redis.conf配置文件redis cluster部分

cluster-enabled yes：是否开启cluster集群

cluster-config-file nodes-6379.conf

集群配置文件，这个文件是redis运行时自动生成的，无须手动配置，也不需要更改

cluster-node-timeout 15000：集群超时时间，即节点超过多久就认为是宕机了

cluster-slave-validity-factor 10

cluster-migration-barrier 1

cluster-require-full-coverage yes：槽是否全覆盖，默认是yes，只要有结点宕机导致16384个槽没全被覆盖，整个集群就全部停止服务，所以一定要改为no

## redis-trib管理器

redis-trib依赖Ruby和RubyGems，以及redis扩展

以下请在root账户下操作

### 安装ruby

如果linux服务器可以联网的话，直接yum安装就好了：yum install ruby

安装ruby的交互工具：**yum install ruby-irb**

### 安装RubyGems

**下载地址：https://github.com/rubygems/rubygems/releases/download/v2.2.3/rubygems-2.2.3.tgz**

**解压缩：tar –zxvf rubygems-2.3.2.tgz**

**cd rubygems-2.3.2**

**安装：ruby setup.rb --no-rdoc --no-ri**

### 安装redis扩展

**下载地址：https://rubygems.org/downloads/redis-3.2.1.gem**

**在所在目录：gem install redis-3.2.1.gem --local --no-rdoc --no-ri**

## 创建集群

命令

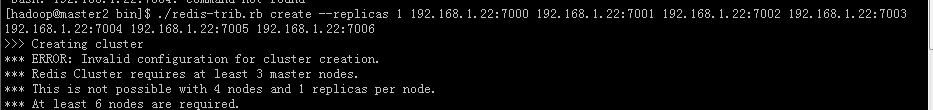
**./redis-trib.rb create --replicas 1 192.168.1.22:7000 192.168.1.22:7001 192.168.1.22:7002 192.168.1.22:7003 192.168.1.22:7004 192.168.1.22:7005 192.168.1.22:7006**

**第一次启动出错：**

****

**原因：redis配置了密码**

**第二次启动出错：**

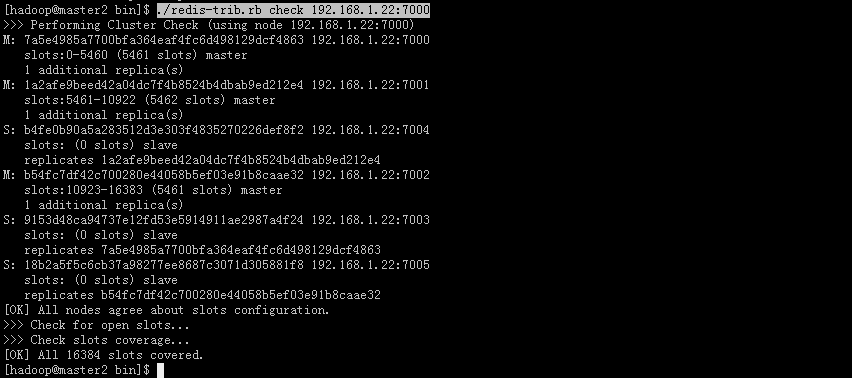
****

**原因：构建集群的命令写错了，应该是：**

**redis-trib.rb create --replicas 1 192.168.1.22:7000 192.168.1.22:7001 192.168.1.22:7002 192.168.1.22:7003 192.168.1.22:7004 192.168.1.22:7005 192.168.1.22:7006**

## 查看集群状态

**./redis-trib.rb check 192.168.1.22:7000**

****

## 集群测试

登录客户端：**redis-cli -c -p 7000 -h 192.168.1.22**

**说明：-c开启reidis cluster模式，连接redis cluster节点时候使用**

**-h 指定连接的ip地址**

**-p 指定连接的端口**

**-a 指定连接redis的密码**

**任意登录其中的一个客户端即可**



## master节点操作

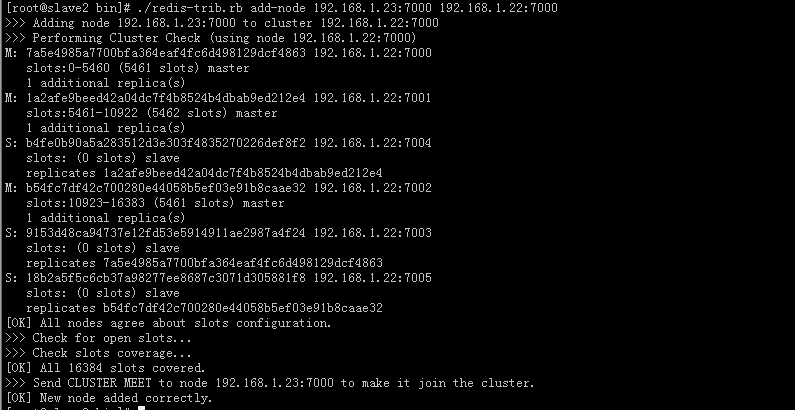
### 添加一个master

1.创建一个新的节点，即一个redis实例192.168.1.23:7000

2.将这个节点node增加到集群中去

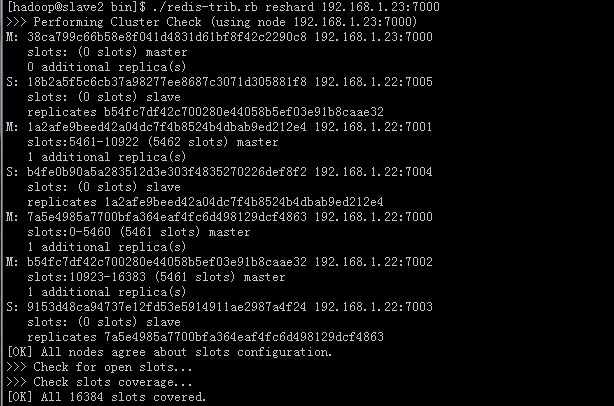
还是使用redis-trib.rb这个脚本, add-node  将一个节点添加到集群里面， 第一个是新节点ip:port, 第二个是任意一个已存在节点ip:port

./redis-trib.rb add-node 192.168.1.23:7000 192.168.1.22:7000

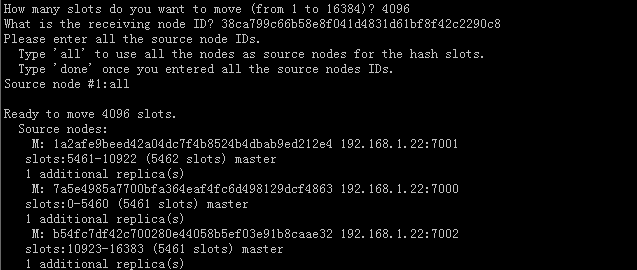


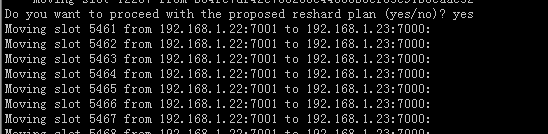
3.给这个node分配slot

./redis-trib.rb reshard 192.168.1.23:7000



根据提示选择要迁移到此node节点的slot数量，比如4096，然后录入当前node的ID，此ID可以在配置文件中查到，或者使用命令查看集群状态获取，然后录入从哪些nodes中提取slot的id，all代表从所有的节点，最后输入yes，开始移动slot





### 删除一个master

删除master节点之前首先要使用reshard移除master的全部slot,然后再删除当前节点(目前只能把被删除master的slot迁移到一个节点上)

1. #把10.10.34.14:6386当前master迁移到10.10.34.14:6380上
2. redis-trib.rb reshard 10.10.34.14:6380
3. #根据提示选择要迁移的slot数量(ps:这里选择500)
4. How many slots **do** you want to move (from 1 to 16384)? 500(被删除master的所有slot数量)
5. #选择要接受这些slot的node-id(10.10.34.14:6380)
6. What is the receiving node ID? c4a31c852f81686f6ed8bcd6d1b13accdc947fd2 (ps:10.10.34.14:6380的node-id)
7. Please enter all the source node IDs.
8. Type 'all' to use all the nodes as source nodes **for** the hash slots.
9. Type 'done' once you entered all the source nodes IDs.
10. Source node #1:f51e26b5d5ff74f85341f06f28f125b7254e61bf(被删除master的node-id)
11. Source node #2:done
12. #打印被移动的slot后，输入yes开始移动slot以及对应的数据.
13. #Do you want to proceed with the proposed reshard plan (yes/no)? yes

然后再删除空的master节点：

./redis-trib.rb del-node 10.10.34.14:6386 'f51e26b5d5ff74f85341f06f28f125b7254e61bf'

[http://static.blog.csdn.net/images/save_snippets_01.png](javascript:;)

## slave节点操作

### 添加一个slave

前2个步骤和matser一样，加入到集群中

然后：./redis-cli –c –p 7000 –h 192.168.1.23

连接上新节点shell,输入命令:cluster replicate 对应master的node-id

注意：note:在线添加slave 时，需要dump整个master进程，并传递到slave，再由 slave加载rdb文件到内存，rdb传输过程中Master可能无法提供服务,整个过程消耗大量io,小心操作

### 删除一个slave

./ redis-trib del-node ip:port '<node-id>'

## slaves扩展reads请求

## 集群命令

1. CLUSTER INFO 打印集群的信息
2. CLUSTER NODES 列出集群当前已知的所有节点（node），以及这些节点的相关信息。
3. 节点
4. CLUSTER MEET <ip> <port> 将 ip 和 port 所指定的节点添加到集群当中，让它成为集群的一份子。
5. CLUSTER FORGET <node\_id> 从集群中移除 node\_id 指定的节点。
6. CLUSTER REPLICATE <node\_id> 将当前节点设置为 node\_id 指定的节点的从节点。
7. CLUSTER SAVECONFIG 将节点的配置文件保存到硬盘里面。
8. 槽(slot)
9. CLUSTER ADDSLOTS <slot> [slot ...] 将一个或多个槽（slot）指派（assign）给当前节点。
10. CLUSTER DELSLOTS <slot> [slot ...] 移除一个或多个槽对当前节点的指派。
11. CLUSTER FLUSHSLOTS 移除指派给当前节点的所有槽，让当前节点变成一个没有指派任何槽的节点。
12. CLUSTER SETSLOT <slot> NODE <node\_id> 将槽 slot 指派给 node\_id 指定的节点，如果槽已经指派给另一个节点，那么先让另一个节点删除该槽>，然后再进行指派。
13. CLUSTER SETSLOT <slot> MIGRATING <node\_id> 将本节点的槽 slot 迁移到 node\_id 指定的节点中。
14. CLUSTER SETSLOT <slot> IMPORTING <node\_id> 从 node\_id 指定的节点中导入槽 slot 到本节点。
15. CLUSTER SETSLOT <slot> STABLE 取消对槽 slot 的导入（**import**）或者迁移（migrate）。
16. 键
17. CLUSTER KEYSLOT <key> 计算键 key 应该被放置在哪个槽上。
18. CLUSTER COUNTKEYSINSLOT <slot> 返回槽 slot 目前包含的键值对数量。
19. CLUSTER GETKEYSINSLOT <slot> <count> 返回 count 个 slot 槽中的键。

## spring-data-redis操作集群

版本说明：

Spring 要4.3.2

spring-data-redis要1.7.2即以上版本

jedis用2.9.0版本

可用版本：

Spring 4.3.2

spring-data-redis 1.7.4

jedis 2.7.2

第一次启动报错：

Jedis does not support password protected Redis Cluster configurations!

原因：

jedisConnectionFactory Bean中配置了

<property name="password" value="redis"></property>

而我的集群没有设置密码

Redis从3.2开始加强安全管理，如果redis没有设置密码，那么redis客户端只能从本地进行访问，如果是从其他机器连接过来访问的，就会报错误。

解决方案是设置参数protected-mode no，这个参数可以动态设置，或者为redis设置密码。

在redis.conf文件中，有个protected-mode no配置，改成no即可

第二次启动报错：加载不了bean redisTemplate

nested exception is java.lang.NoSuchMethodError: org.springframework.core.serializer.support.DeserializingConverter.<init>(Ljava/lang/ClassLoader;)V

原因：版本不兼容，改成上述jar正常

## 集群方案：

### 哨兵-Sentinel

### Codis

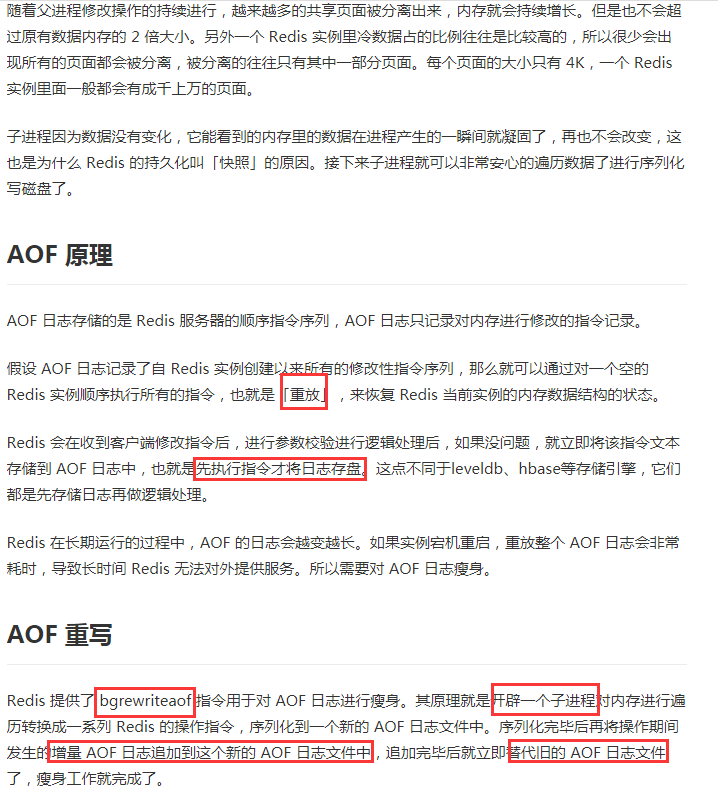
### Redis Cluster

# 持久化

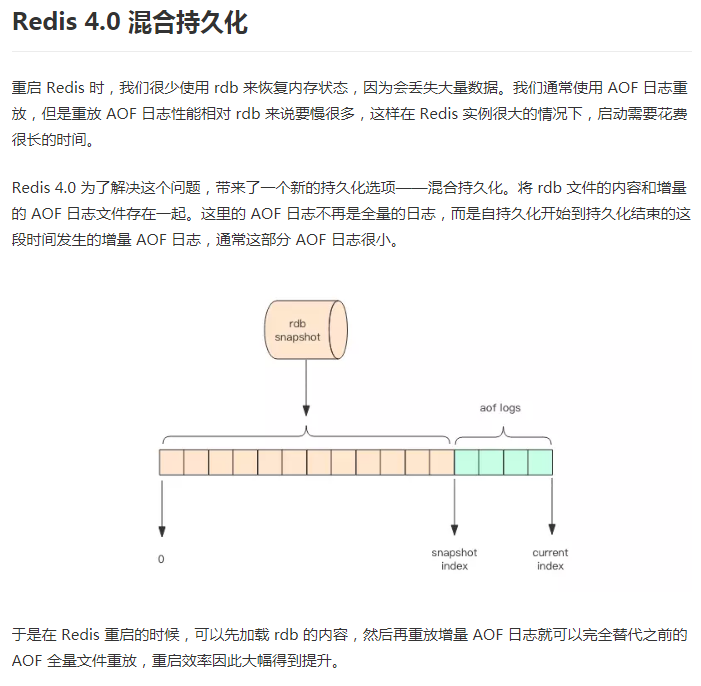
AOF和ADB











## Fork

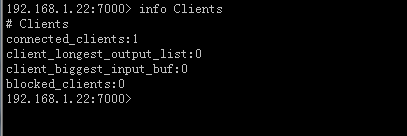


# Redis常用运维命令

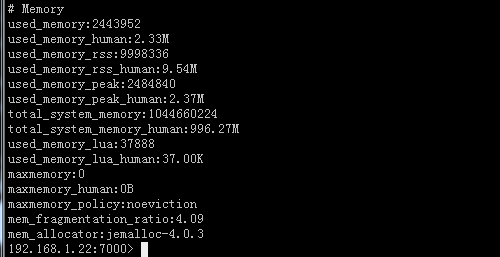
**info server**

****

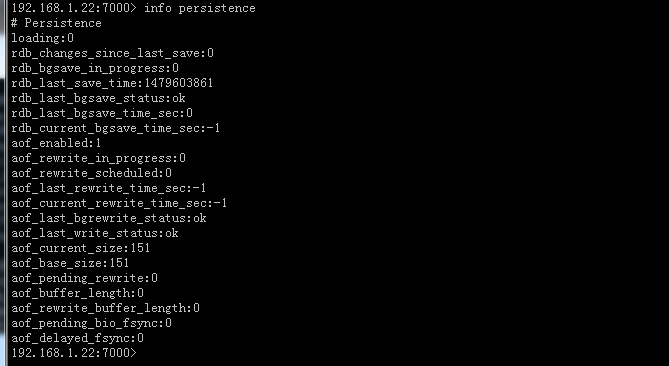
**info clients**

****

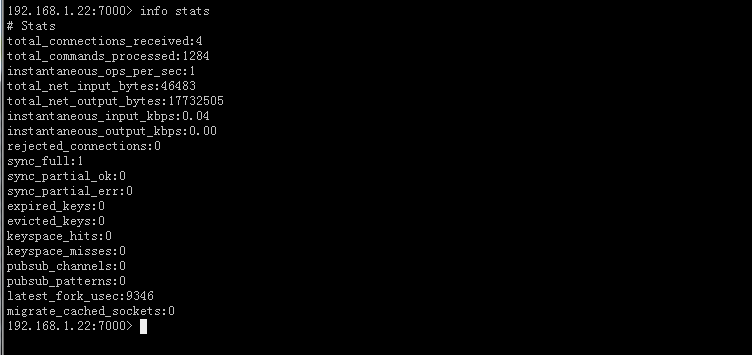
**info memory**

****

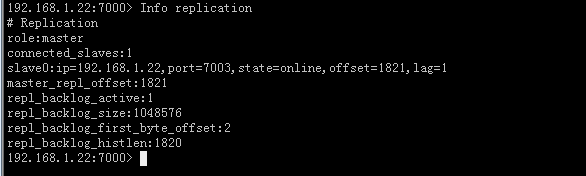
**info persistence:记录redis的RDB和AOF持久化信息**

****

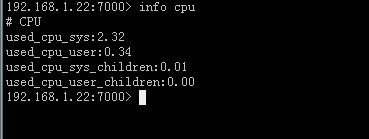
**Info stats**

****

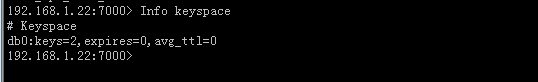
**Info replication**

****

**Info cpu**

****

**Info keyspace**

****

# redis存储tomcat的session

## 在tomcat的service.xml中配置

1、单点redis

<Context path="" docBase="/home/tomcat/yghshwx">

<Valve className="com.orangefunction.tomcat.redissessions.RedisSessionHandlerValve" />

<Manager className="com.orangefunction.tomcat.redissessions.RedisSessionManager"

host="10.46.66.13"

port="6379"

database="2"

maxInactiveInterval="60"

/>

</Context>

2、哨兵模式Sentinel集群

<Valve className="com.orangefunction.tomcat.redissessions.RedisSessionHandlerValve" />

<Manager className="com.orangefunction.tomcat.redissessions.RedisSessionManager"

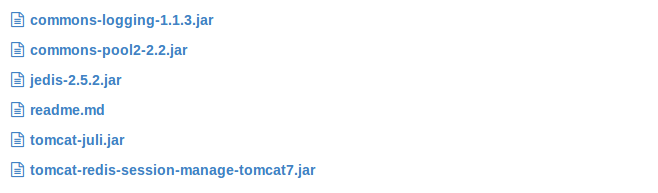
maxInactiveInterval="60"

sentinelMaster="mymaster"

sentinels="127.0.0.1:26379,127.0.0.1:26380,127.0.0.1:26381,127.0.0.1:26382"/>

3、cluster分布式集群配置

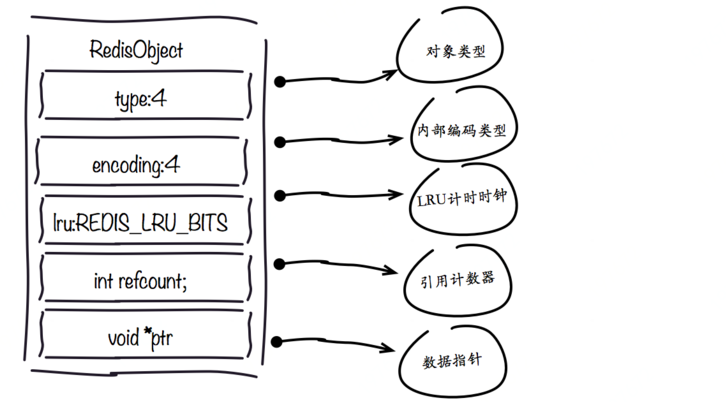
## 需要的jar包



# 优化

## redisObject对象

Redis存储的所有值对象在内部定义为redisObject结构体，内部结构如下图所示



## 降低内存占用

1、短结构

这个通过配置就可以实现，配置文件中的高级配置

原理：

改变了key和value的内部编码结构

示例：有个100万键值对的hash散列

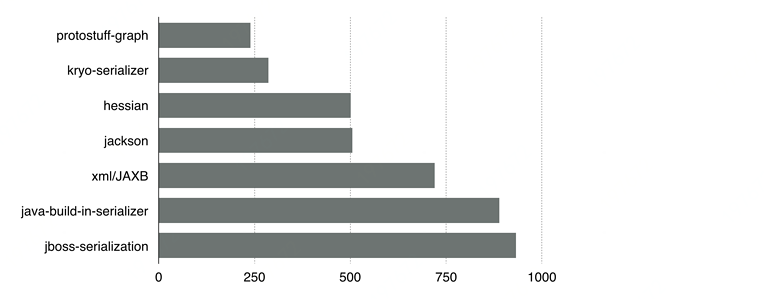
不要放到一个散列中，我们可以把每100个子key放到一个散列中，在使用短结构（压缩列表），这样会节省很大内存。

## 缩减键值对象

降低Redis内存使用最直接的方式就是缩减键（key）和值（value）的长度。

key长度：如在设计键时，在完整描述业务情况下，键值越短越好。

value长度：值对象缩减比较复杂，常见需求是把业务对象序列化成二进制数组放入Redis。首先应该在业务上精简业务对象，去掉不必要的属性避免存储无效数据。其次在序列化工具选择上，应该选择更高效的序列化工具来降低字节数组大小。以JAVA为例，内置的序列化方式无论从速度还是压缩比都不尽如人意，这时可以选择更高效的序列化工具，如: protostuff，kryo等，下图是JAVA常见序列化工具空间压缩对比。



其中java-built-in-serializer表示JAVA内置序列化方式，更多数据见jvm-serializers项目: https://github.com/eishay/jvm-serializers/wiki，其它语言也有各自对应的高效序列化工具。

值对象除了存储二进制数据之外，通常还会使用通用格式存储数据比如:json，xml等作为字符串存储在Redis中。这种方式优点是方便调试和跨语言，但是同样的数据相比字节数组所需的空间更大，在内存紧张的情况下，可以使用通用压缩算法压缩json,xml后再存入Redis，从而降低内存占用，例如使用GZIP压缩后的json可降低约60%的空间。

开发提示：当频繁压缩解压json等文本数据时，开发人员需要考虑压缩速度和计算开销成本，这里推荐使用google的Snappy压缩工具，在特定的压缩率情况下效率远远高于GZIP等传统压缩工具，且支持所有主流语言环境。

## 共享对象池

注意：开启maxmemory和LRU淘汰策略后对象池无效

# 内部编码

## ziplist（压缩列表）

压缩列表的优点：

压缩列表ziplist结构本身就是一个连续的内存块，由表头、若干个entry节点和压缩列表尾部标识符zlend组成，通过一系列编码规则，提高内存的利用率，使用于存储整数和短字符串。

压缩列表的缺点：

每次插入或删除一个元素时，都需要进行频繁的调用realloc()函数进行内存的扩展或减小，然后进行数据”搬移”，甚至可能引发连锁更新，造成严重效率的损失。

## linkedlist（链表）

当列表类型无法满足ziplist要求时，redis会采用linkedlist做为列表的内部实现。

# 大促期间热点数据解决方案

## Redis

没有很好的解决方案

## Tair

参考：

https://segmentfault.com/a/1190000012636708

阿里巴巴开源的弹性缓存/存储平台

解决方案：分为三部分：热点识别、读热点方案和写热点方案

### 热点统计过程

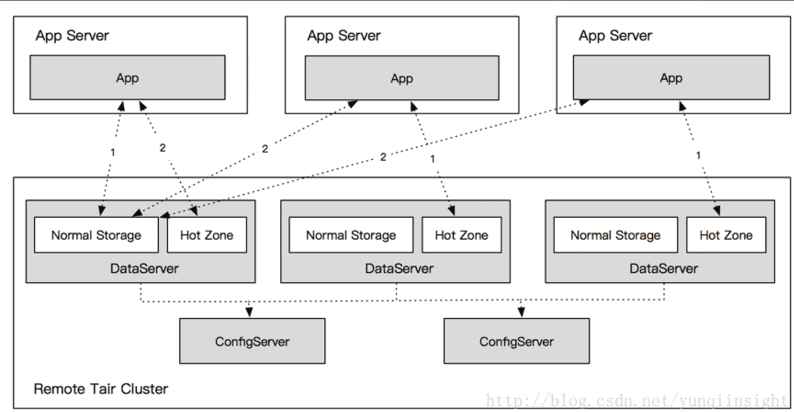
DataServer收到客户端的请求后，由每个具体处理请求的工作线程（Worker Thread）进行请求的统计。工作线程用来统计热点的数据结构均为ThreadLocal模式的数据结构，完全无锁化设计。热点识别算法使用精心设计的多级加权LRU链和HashMap组合的数据结构，在保证服务端请求处理效率的前提下进行请求的全统计，支持QPS热点和流量热点（即请求的QPS不大但是数据本身过大而造成的大流量所形成的热点）的精准识别。每个采样周期结束时，工作线程会将统计的数据结构转交到后台的统计线程池进行分析处理。统计工作异步在后台进行，不抢占正常的数据请求的处理资源。

### 读热点方案

#### 服务端设计

原始Tair的数据访问方式是先进行Hash(Key)%BucketCount的计算，得出具体的数据存储Bucket，再检索数据路由表找到该Bucket所在的DataServer后对其进行读写请求的。所以相同Key的读写请求必然落在固定的DataServer上，且无法通过水平扩展DataServer数量来解决。

本方案通过在DataServer上划分一块HotZone存储区域的方式来解决热点数据的访问。该区域存储当前产生的所有读热点的数据，由客户端配置的缓存访问逻辑来处理各级缓存的访问。多级缓存架构如下：



所有DataServer的HotZone存储区域之间没有权重关系，每个HotZone都存储相同的读热点数据。客户端对热点数据Key的请求会随机到任意一台DataServer的HotZone区域，这样单点的热点请求就被散列到多个节点乃至整个集群。

#### 客户端设计

当客户端在第一次请求前初始化时，会获取整个Tair集群的节点信息以及完整的数据路由表，同时也会获取配置的热点散列机器数（即客户端访问的HotZone的节点范围）。随后客户端随机选择一个HotZone区域作为自身固定的读写HotZone区域。在DataServer数量和散列机器数配置未发生变化的情况下，不会改变选择。即每个客户端只访问唯一的HotZone区域。

客户端收到服务端反馈的热点Key信息后，至少在客户端生效N秒。在热点Key生效期间，当客户端访问到该Key时，热点的数据会首先尝试从HotZone节点进行访问，此时HotZone节点和源数据DataServer节点形成一个二级的Cache模型。客户端内部包含了两级Cache的处理逻辑，即对于热点数据，客户端首先请求HotZone节点，如果数据不存在，则继续请求源数据节点，获取数据后异步将数据存储到HotZone节点里。使用Tair客户端的应用常规调用获取数据的接口即可，整个热点的反馈、识别以及对多级缓存的访问对外部完全透明。HotZone缓存数据的一致性由客户端初始化时设置的过期时间来保证，具体的时间由具体业务对缓存数据不一致的最大容忍时间来决定。

客户端存储于本地的热点反馈过期后，数据Key会到源DataServer节点读取。如果该Key依旧在服务端处于热点状态，客户端会再次收到热点反馈包。因为所有客户端存储于本地的热点反馈信息的失效节奏不同，所以不会出现同一瞬间所有的请求都回源的情况。即使所有请求回源，也仅需要回源读取一次即可，最大的读取次数仅为应用机器数。若回源后发现该Key已不是热点，客户端便回到常规的访问模式。

### 写热点方案

写热点的方案对客户端完全透明，不需要客户端做任何修改。

# 指定前缀的key

[Jedis scan及其count的值](https://blog.csdn.net/zy_281870667/article/details/79310554)

## Keys命令与Scan命令

@Test  
public void scanRedis(){  
 ReloadableJimClientFactory factory = new ReloadableJimClientFactory();  
  
 factory.setJimUrl("jim://1803528671997086613/2");  
  
 Cluster client = factory.getClient();  
 ScanOptions.ScanOptionsBuilder scanOptionBuilder = ScanOptions.*scanOptions*();  
  
 scanOptionBuilder.count(100);  
 scanOptionBuilder.match("airplane\_empty\_search\_whitelist\*");  
 ScanOptions scanOption = scanOptionBuilder.build();  
 KeyScanResult<String> scan = client.scan(null, scanOption);  
  
 while (!scan.isFinished()) {  
 scan = client.scan(scan.getCursor(), scanOption);  
  
 List<String> result = scan.getResult();  
  
 System.*out*.println("scan result:" + result);  
 if(result.size()>0){  
 System.*out*.println("scan result value:" + jdCacheUtils.hGetAll(result.get(0) ));  
 }  
 }  
}

KEYS命令一次性返回所有匹配的key，所以，当redis中的key非常多时，对于内存的消耗和redis服务器都是一个隐患，

对于Redis 2.8以上版本给我们提供了一个更好的遍历key的命令 SCAN 该命令的基本格式：

SCAN cursor [MATCH pattern] [COUNT count]

SCAN 每次执行都只会返回少量元素，所以可以用于生产环境，而不会出现像 [KEYS](http://www.redis.cn/commands/keys.html) 或者 [SMEMBERS](http://www.redis.cn/commands/smembers.html) 命令带来的可能会阻塞服务器的问题。

[SCAN](http://www.redis.cn/commands/scan.html)命令是一个基于游标的迭代器。这意味着命令每次被调用都需要使用上一次这个调用返回的游标作为该次调用的游标参数，以此来延续之前的迭代过程

当[SCAN](http://www.redis.cn/commands/scan.html)命令的游标参数（即cursor）被设置为 0 时， 服务器将开始一次新的迭代， 而当服务器向用户返回值为 0 的游标时， 表示迭代已结束。

## COUNT选项

对于增量式迭代命令不保证每次迭代所返回的元素数量，我们可以使用**COUNT**选项， 对命令的行为进行一定程度上的调整。COUNT 选项的作用就是让用户告知迭代命令， 在每次迭代中应该从数据集里返回多少元素。使用COUNT 选项对于对增量式迭代命令相当于一种提示， 大多数情况下这种提示都比较有效的控制了返回值的数量。

**注意:**COUNT选项并不能严格控制返回的key数量，只能说是一个大致的约束。并非每次迭代都要使用相同的 COUNT 值，用户可以在每次迭代中按自己的需要随意改变 COUNT 值， 只要记得将上次迭代返回的游标用到下次迭代里面就可以了

## MATCH 选项

类似于KEYS 命令，增量式迭代命令通过给定 MATCH 参数的方式实现了通过提供一个 glob 风格的模式参数， 让命令只返回和给定模式相匹配的元素。

MATCH 选项对元素的模式匹配工作是在命令从数据集中取出元素后和向客户端返回元素前的这段时间内进行的， 所以如果被迭代的数据集中只有少量元素和模式相匹配， 那么迭代命令或许会在多次执行中都不返回任何元素。

## Pipeline批量操作

[Redis中PipeLine使用](https://blog.csdn.net/zmx729618/article/details/52681386)

@Test  
public void testPipeline() throws Exception{  
 ReloadableJimClientFactory factory = new ReloadableJimClientFactory();  
 factory.setJimUrl("jim://1803528671997086613/2");  
  
 Cluster client = factory.getClient();  
 PipelineClient pipeline = client.pipelineClient();  
  
 try{  
 //没有使用pipeline  
 long start=System.*currentTimeMillis*();  
 for (int i = 0; i <500; i++) {  
 jdCacheUtils.set(String.*valueOf*(i),String.*valueOf*(i));  
 }  
 long end=System.*currentTimeMillis*();  
 System.*out*.println("the total time is:"+(end-start));  
  
 //使用pipeline去set  
 long start\_pipe=System.*currentTimeMillis*();  
 for (int i = 0; i <500; i++) {  
 pipeline.set(String.*valueOf*(i).getBytes(),("kil"+i).getBytes());  
 }  
 List<Object> result2 = pipeline.flushAndReturnAll();  
 long end\_pipe=System.*currentTimeMillis*();  
 System.*out*.println("the pipe total time is:"+(end\_pipe-start\_pipe));  
  
 //使用pipeline去get  
 long start\_pipe1=System.*currentTimeMillis*();  
 for (int i = 0; i <500; i++) {  
 pipeline.get(String.*valueOf*(i).getBytes());  
 }  
 List<Object> result3 = pipeline.flushAndReturnAll();  
 long end\_pipe1=System.*currentTimeMillis*();  
 System.*out*.println("the pipe get total time is:"+(end\_pipe1-start\_pipe1));  
  
 //将get到的结果字节数组转换成字符串  
 System.*out*.println("pipeline get: " + result3.size());  
 for(int i=0; i<result3.size(); i++){  
 byte[] bytes = (byte[])result3.get(i);  
 System.*out*.println("pipeline get size: " + new String(bytes, "utf-8"));  
 }  
  
 pipeline = client.pipelineClient();  
  
 pipeline.sMembers("air\_base\_rule\_prom\_airline\_act\_HGH\_XIY".getBytes());  
 pipeline.sAdd("air\_base\_rule\_prom\_airline\_act\_HGH\_XIY".getBytes(), "huyanxia".getBytes());  
 pipeline.exists("air\_base\_rule\_prom\_airline\_act\_HGH\_XIY".getBytes());  
  
 //pipeline.sMembers("air\_base\_rule\_prom\_airline\_act\_HGH\_XIY".getBytes());  
  
 List<Object> result4 = pipeline.flushAndReturnAll();  
 Set<byte[]> bytes = (Set<byte[]>) result4.get(0);  
  
 for (byte[] bytes1 : bytes) {  
 System.*out*.println("pipeline:" + new String(bytes1, "utf-8"));  
 }  
  
 System.*out*.println("pipeline:" + result4);  
 System.*out*.println("pipeline:" + result4.get(0).toString());  
 }finally{  
 if(pipeline!=null){  
 pipeline.close();  
 }  
 }  
}

一般情况下，Redis Client端发出一个请求后，通常会阻塞并等待Redis服务端处理，Redis服务端处理完后请求命令后会将结果通过响应报文返回给Client。

这有点类似于HBase的Scan，通常是Client端获取每一条记录都是一次RPC调用服务端。

在Redis中，有没有类似HBase Scanner Caching的东西呢，一次请求，返回多条记录呢？

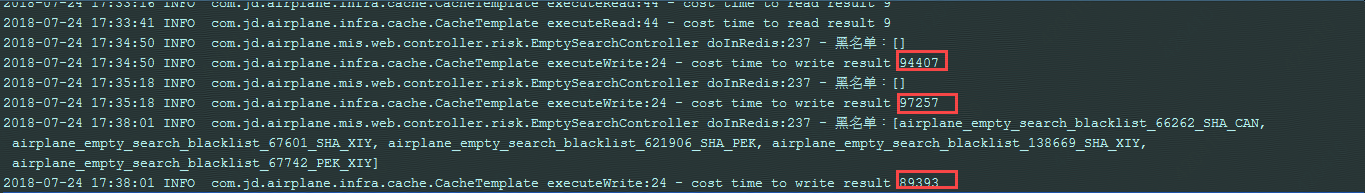
有，这就是Pipline。官方介绍 http://redis.io/topics/pipelining

通过pipeline方式当有大批量的操作时候，我们可以节省很多原来浪费在网络延迟的时间，需要注意到是用pipeline方式打包命令发送，redis必须在处理完所有命令前先缓存起所有命令的处理结果。打包的命令越多，缓存消耗内存也越多。所以并不是打包的命令越多越好。

使用Pipeline在对Redis批量读写的时候，性能上有非常大的提升。

## 耗时

太慢：测试环境耗时80秒



生产环境也非常慢



# Zset

Zset跟Set之间可以有并集运算，因为他们存储的数据字符串集合，不能有一样的成员出现在一个zset中，但是为什么有了set还要有zset呢？zset叫做有序集合，而set是无序的，zset怎么做到有序的呢？就是zset的每一个成员都有一个分数与之对应，并且分数是可以重复的。有序集合的增删改由于有啦排序，执行效率就是非常快速的，即便是访问集合中

间的数据也是非常高效的。

[Redis学习手册(Sorted-Sets数据类型)](https://www.cnblogs.com/stephen-liu74/archive/2012/03/23/2354994.html)

[Redis数据结构详解之Zset（五）](https://www.cnblogs.com/knowledgesea/p/4999288.html)

| **参数** | **解释** |
| --- | --- |
| Zadd | Zadd key-name score member [score member…]——将带有给定分值的成员添加到有序列表里面 |
| Zrem | Zrem key-name member [member…]——从有序集合里面移除给定的成员，并返回被移除成员的数量 |
| Zcard | Zcard key-name——返回有序集合包含的成员数量 |
| Zincrby | Zincrby key-name increment member——将member成员的分值加上increment |
| Zcount | Zcount key-name min max——返回分值介于min和max之间的成员数量，包括min和max在内 |
| Zrank | Zrank key-name member——返回成员member在有序集合中的排名，成员按照分值从小到大排列 |
| Zrevrank | Zrevrank key-name member——返回成员member在有序集合中的排名 ，成员按照分值从大到小排列 |
| Zscore | Zscore key-name member——返回成员member的分值 |
| Zrange | Zrange key-name start stop [WITHSCORES]——返回有序集合中排名介于start和stop之间的成员，包括start和stop在内，如果给定了可选的WITHSCORES选项，那么命令会将成员的分值一并返回，成员按照分值从小到大排列 |
| Zrevrange | Zrevrange key-name start stop [WITHSCORES]——返回有序集合中排名介于start和stop之间的成员，包括start和stop在内，如果给定了可选的WITHSCORES选项，那么命令会将成员的分值一并返回，成员按照分值从大到小排列 |
| Zrangebyscore | Zrangebyscore key-name min max [WITHSCORES] [LIMIT offset count]——返回有序集合中分值介于min和max之间的所有成员，包括min和max在内，并按照分值从小到大的排序来返回他们 |
| Zrevrangebyscore | Zrevrangebyscore key-name min max [WITHSCORES] [LIMIT offset count]——返回有序集合中分值介于min和max之间的所有成员，包括min和max在内，并按照分值从大到小的排序来返回他们 |
| Zremrangebyrank | Zremrangebyrank key-name start stop——移除有序集合中排名介于start和stop之间的所有成员，包括start和stop在内 |
| Zremrangebyscore | Zremrangebyscore key-name key min max——移除有序集合中分值介于min和max之间的所有成员，包括min和max在内 |
| Zinterstore | Zinterstore dest-key key-count key [key…][WEIGHTS weight [weight…]] [AGGREGATE SUM|MIN|MAX]——对给定的有序集合执行类似于集合的交集运算 |
| Zunionstore | Zunionstore dest-key key-count key [key…][WEIGHTS weight [weight…]] [AGGREGATE SUM|MIN|MAX]——对给定的有序集合执行类似于集合的并集运算 |

# Redis生存周期

*、在 Redis 中，带有生存时间的 key 被称为『易失的』(volatile)  
\* 生存时间可以通过使用 DEL 命令来删除整个 key 来移除，或者被 SET 和 GETSET 命令覆写(overwrite)，这意味着，  
\* 如果一个命令只是修改(alter)一个带生存时间的 key 的值而不是用一个新的 key 值来代替(replace)它的话，那么生存时间不会被改变。  
\* 比如说，对一个 key 执行 INCR 命令，对一个列表进行 LPUSH 命令，或者对一个哈希表执行 HSET 命令，这类操作都不会修改 key 本身的生存时间。  
\* 另一方面，如果使用 RENAME 对一个 key 进行改名，那么改名后的 key 的生存时间和改名前一样。  
\* RENAME 命令的另一种可能是，尝试将一个带生存时间的 key 改名成另一个带生存时间的 another\_key ，  
\* 这时旧的 another\_key (以及它的生存时间)会被删除，然后旧的 key 会改名为 another\_key ，因此，新的 another\_key 的生存时间也和原本的 key 一样。  
\* 使用 PERSIST 命令可以在不删除 key 的情况下，移除 key 的生存时间，让 key 重新成为一个『持久的』(persistent) key 。  
\* 2、更新生存时间  
\* 可以对一个已经带有生存时间的 key 执行 EXPIRE 命令，新指定的生存时间会取代旧的生存时间。  
\* 3、过期时间的精确度  
\* 在 Redis 2.4 版本中，过期时间的延迟在 1 秒钟之内 —— 也即是，就算 key 已经过期，但它还是可能在过期之后一秒钟之内被访问到，  
\* 而在新的 Redis 2.6 版本中，延迟被降低到 1 毫秒之内。*

# 项目中实践

## 促销数据的Redis缓存设计

### 原数据结构

1. Hash散列：命令hSet

hset key field value

field 不存在，直接存储。

field 存在，执行覆盖操作。

Key：

air\_base\_prom\_activity\_sku

Filed:

venderId\_flightCode\_clazzCode,如："621906\_FM9531\_B"

Value：

{

"activityId": 1800,

"clazzCode": "B",

"clazzPrice": 300,

"createDate": 1497602389000,

"flightCode": "FM9531",

"id": 11609147,

"modifiedDate": 1497602389000,

"promotionBeans": 0,

"promotionPrice": 280,

"reducePrice": 20,

"rfId": 4886089506,

"skuId": 200105677302,

"status": "1",

"userType": "0",

"venderId": 621906,

"cacheFiled": "621906\_FM9531\_B"

}

1. 问题：

Key下面的subKey太多，可能有100万左右。这样设计不合理

### 新数据结构

#### 提供给前台

将key打散（在不同分片上），存储成Object关联到key，底层实现原理是set(key, JSON.toJSONString(value)),

这样key下面value值少，方便get

缺点：需要序列化及反序列化

1. String：命令Set

Set(key，value)

Key：air\_base\_rule\_prom\_act\_sku\_venderId\_flightCode\_clazzCode，

如：" air\_base\_rule\_prom\_act\_sku\_621906\_FM9531\_B"

Key加前缀的目的，是怕读缓存乱。

Value：{

"id": 11609147,

"activityId": 1800, 活动id

" rfId": 购物车id 下单时用

" skuId": skuId 下单时用

"price": 300, 票面价

"reducePrice": 20, 立减金额

"Type": "0", 促销类型

"Beans": 0, 京豆数量

"Key": "621906\_FM9531\_B" 缓存key

}

#### 关联缓存

结构：set集合

命令：sadd 、smember、scard、sismember

key：relation

value：商家\_航司

#### 商家\_航司缓存

结构：set集合

命令：sadd 、smember、scard、sismember

key：商家\_航司

value：venderId\_flightCode\_clazzCode

### 缓存时间

若设置缓存过期时间，那么worker就要每天都跑全量的数据，耗时大。

因此缓存无过期时间，永久有效。每天晚上跑worker，删除失效的数据。白天通过消费MQ消息更新增加的数据。

## 空查询

### 缓存结构

#### 方案一：

黑名单和白名单使用一个key存储，另外空查询次数累计使用一个key，用于批量查询的关联key

黑白名单

结构：String类型，setEx，设置缓存过期时间

key：air\_base\_rule\_empty\_search\_risk\_66262\_PEK\_SHA

value: EmptySearchEntity对象的Json串 {"arrAirportCode":"SHA","depAirportCode":"PEK","type":"WHITE","venderId":"66262"}

空查询次数

结构：String类型，setEx，设置缓存过期时间，与黑名单过期时间一样

key：

air\_base\_rule\_empty\_search\_count\_66262\_PEK\_SHA

value：1

关联：

结构：set类型，sAdd，定期删除缓存

Key: air\_base\_rule\_empty\_search\_relation\_66262

Value: 66262\_PEK\_SHA, 66262\_CTU\_SHA

#### 方案二：

方案一中黑名单和白名单耦合在一起，这样批量查询时耗时过长。因为用户关心的是黑名单，所以黑名单查询的次数最多，而且黑名单中的个数比白名单少（自营的4000多在白名单中）。而且没办法分页查询，关联key是set结构，无序的。

因此将黑名单和白名单分开存储，而且关联key使用zSet结构（有序）。

黑名单：

结构：String类型，setEx，设置缓存过期时间

Key：*air\_base\_rule\_empty\_search\_blacklist\_venderId\_depAirportCode\_arrAirportCode*

Value: EmptySearchEntity对象的Json串 {"arrAirportCode":"SHA","depAirportCode":"PEK","type":"BLACK","venderId":"66262"}

白名单：

结构：String类型，setEx，设置缓存过期时间

Key：*air\_base\_rule\_empty\_search\_whitelist\_venderId\_depAirportCode\_arrAirportCode*

Value: EmptySearchEntity对象的Json串 {"arrAirportCode":"SHA","depAirportCode":"PEK","type":"WHITE","venderId":"66262"}

空查询次数

结构：String类型，setEx，设置缓存过期时间，与黑名单过期时间一样

key：

air\_base\_rule\_empty\_search\_count\_66262\_PEK\_SHA

value：1

白名单关联：

结构：zset类型，zAdd，定期删除缓存

Key: air\_base\_rule\_empty\_search\_whitelist\_relation\_66262

Value: 66262\_PEK\_SHA, 66262\_CTU\_SHA

Score: 过期时间的时间戳

黑名单关联：

结构：zset类型，zAdd，定期删除缓存

Key: air\_base\_rule\_empty\_search\_blacklist\_relation\_66262

Value: 66262\_PEK\_SHA, 66262\_CTU\_SHA

Score: 过期时间的时间戳

### **查询**

#### 批量查询

使用scan命令结合pipeline，但是耗时非常大。Scan命令遍历指定前缀的key，会搜索所有的机器中的所有分片

#### **分页查询**

目前分页的方式有2种

1. 从redis拿出所有数据后，再做内存分页（不推荐），热点数据小的时候可以这样做，性能相差不是很大，但是当数据量大的时候，分页期间就会占用大量内存，或撑爆；
2. 基于redis的数据结构做缓存分页，这里又分2种

①：基于redis的list数据结构，直接通过list的数据结构，用range方法可以进行分页，在数据量大的时候，性能也很可观，但是当存在接口高并发访问时，这个list可能会无限延长，且里面的数据会存在很多重复，这就会影响到正常的业务（不是很推荐）；

②：基于redis的ZSet数据结构，通过Zset这个有序集合我们也可以做分页，同样也是用range方法，但是这里比较麻烦的是在初始化数据的时候Zset必须存放TypedTuple类型的数据，这个类型是一个value和score的键值对，具体可以查百度，这个score的生成比较麻烦我这边测试时用的是当前数据在这个list的位置，然后Zset是根据这个score值来排序的，默认是从小到大；用这个的好处是，即使在高并发情况下Zset中也不会存在重复数据从而影响正常的业务；而且分页效率也和list结构差不多；

③：用hash和Zset来一起实现；这个是问了一个朋友和得知的，Zset中存储有序的id字段，通过分页后拿到id，然后再用id去hash中取，感觉应该效率相差不大的，只是中间多了层从hash结构取，还需要维护又一个hash；（为何这样做我也不清楚）；

#### 项目中使用

使用zSet结构，score是时间戳。

航线下查询时分页，通过关联key查出符合score条件的pageSize条数据。

[Redis有序集合命令ZRANGEBYSCORE|ZREVRANGEBYSCORE|ZREMRANGEBYRANK|ZREMRANGEBYSCORE](https://blog.csdn.net/chengqiuming/article/details/79184906)

Set<String> prefix = cluster.zRevRangeByScore(relationKey, 0, new Date().getTime(), (pageNo - 1) \* pageSize, pageSize);

Set<String> prefix = cluster.zRevRangeByScore(relationKey, new Date().getTime(), Double.*MAX\_VALUE*, (pageNo - 1) \* pageSize, pageSize);

[**基于redis的延迟消息队列设计**](https://www.cnblogs.com/peachyy/p/7398430.html)

# 缓存击穿问题

案例：

当有10000万个线程同时访问redis的key（相同key），当key过期或者缓存数据丢失时，10000万个线程同时取访问数据区。这种情况成为缓存击穿问题。

解决：

当出现key过期时，可以在redis和数据库之间设置分布式锁，使只有一个线程取访问数据库，拿到数据刷新缓存，其他的9999个线程阻塞，当第一个线程刷缓存成功后，然后9999个线程可以访问缓存

# redis的线程安全问题

## Redis不存在线程安全问题？

Redis采用了线程封闭的方式，把任务封闭在一个线程，自然避免了线程安全问题，不过对于需要依赖多个redis操作的复合操作来说，依然需要锁，而且有可能是分布式锁。

复合操作：get、set等多个命令

[单进程单线程的Redis如何能够高并发](https://blog.csdn.net/liupeng_qwert/article/details/77263187)

## 分布式锁

[分布式锁](https://mp.weixin.qq.com/s/8fdBKAyHZrfHmSajXT_dnA)

[Redis分布式锁的正确实现方式](https://www.cnblogs.com/linjiqin/p/8003838.html)

加锁：

方法一：jedis.set(String key, String value, String nxxx, String expx, int time)

* 第一个为key，我们使用key来当锁，因为key是唯一的。
* 第二个为value，我们传的是requestId，很多童鞋可能不明白，有key作为锁不就够了吗，为什么还要用到value？原因就是我们在上面讲到可靠性时，分布式锁要满足第四个条件解铃还须系铃人，通过给value赋值为requestId，我们就知道这把锁是哪个请求加的了，在解锁的时候就可以有依据。requestId可以使用UUID.randomUUID().toString()方法生成。
* 第三个为nxxx，这个参数我们填的是NX，意思是SET IF NOT EXIST，即当key不存在时，我们进行set操作；若key已经存在，则不做任何操作；
* 第四个为expx，这个参数我们传的是PX，意思是我们要给这个key加一个过期的设置，具体时间由第五个参数决定。
* 第五个为time，与第四个参数相呼应，代表key的过期时间。

方法二：

jedis.setnx()和jedis.expire()组合实现加锁，不是原子操作，线程同步问题

解锁：

方法一：

直接del（），线程同步问题

为了防止线程del其他线程的锁，所以value使用uuid，在del的时候判断是否value为uuid（有可能uuid不唯一，多个线程的uuid一样，所以可以在uuid中加上threadId）

方法二：

先判断是不是当前线程的锁，再del，线程同步问题

方法三：

执行lua脚本，没试过

超时时间>业务的处理时间，或者可以动态延长锁的超时时间（在获得锁后）

### 项目中使用：

风控中，Ip查询日志写入es时，使用redis分布式锁（3秒），若获得锁成功则写入es，若获得锁失败则将日志写入redis缓存中，另外跑worker（多线程）去将缓存中的数据写入es中（使用redis分布式锁（3秒），若获得锁成功则写入es，若获得锁失败则将日志写入redis缓存中）

Ip查询日志写入es时，

trylock(value, "", *CACHE\_TIME\_OUT*, false)

jmi分装了一下，false表示“NX”，默认“PX”

加锁：

private boolean trylock(String key, String value, long timeout, final boolean exist) {  
 return jdCacheUtils.set(key, value, timeout, exist);  
}

释放锁：会存在释放其他线程锁的可能

private boolean unlock(String key) {  
 return jdCacheUtils.del(key);  
}

### 对比

[sharkStore是如何实现更靠谱的分布式锁的？](http://jpcloud.jd.com/pages/viewpage.action?pageId=22587661)

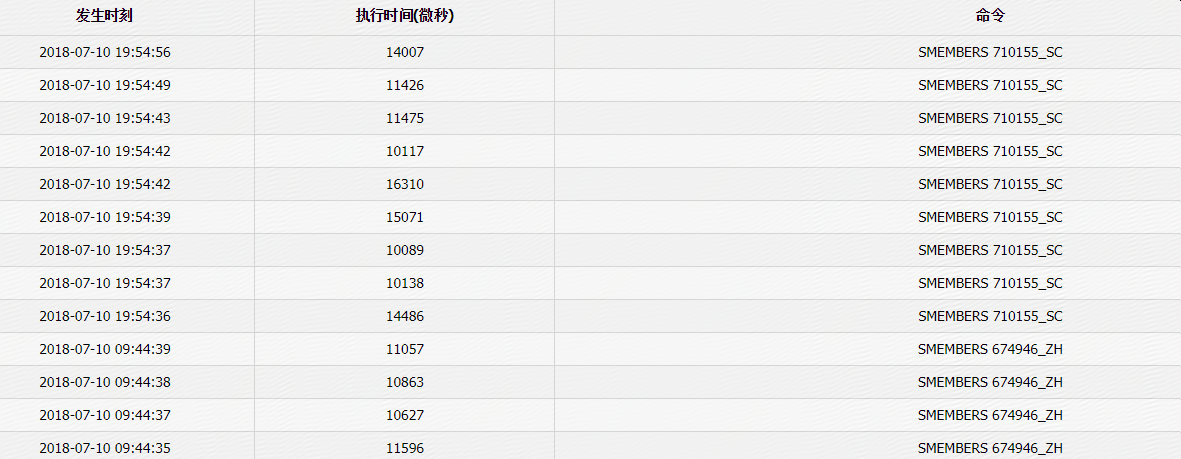
# Redis小记

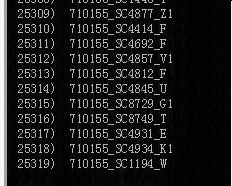
## 是否需要删除key

不需要，只要key下面没有值就不会占用空间

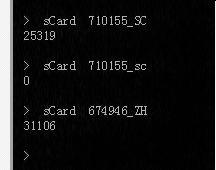
## 慢日志

SMEMBERS 678081\_SC





Redis命令区分大小写，当value值过多时，sMembers比sCard耗时多很多



### Exist耗时

在排查Redis耗时的过程中发现大量的EXISTS命令耗时很长。很意外，印象中感觉这个这个命令应该是很快执行完的。随后了解到

1. Redis中DEL命令的时间复杂度对于字符串和哈希值而言是O(1)，而对于list、set和sorted set而言则是O(N) （这里的 N 是集合中数据项的数目）。所以对于大集合key，DEL命令会很耗时。

2. 在EXISTS命令处理函数中实现了清除过期key的主动策略，会先调用 expireIfNeeded 函数检查要访问的 key 是否过期，如果过期就delete掉这个key。所以对于大集合，EXISTS命令也有可能会很耗时。

### sMember耗时

smembers命令用于获取集合全集，时间复杂度为O(N),N为集合中的数量，set集合数量控制在500以内

[Redis时延问题分析及应对](https://www.cnblogs.com/me115/p/5032177.html)

### 命令时间复杂度

[**Redis基础、常用类型介绍、时间复杂度**](https://blog.csdn.net/zzm848166546/article/details/80360665)

## Jimdb

[java io -- FilterInputStream 与 装饰者模式](https://blog.csdn.net/zhao123h/article/details/52826682)

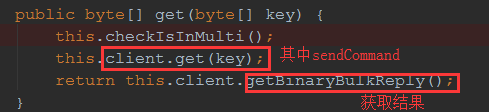
[剖析Redis协议(续)](https://blog.csdn.net/weixin_34149796/article/details/86975691)

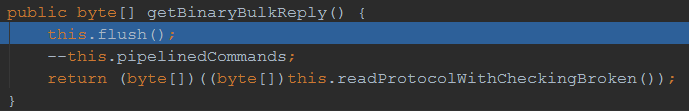


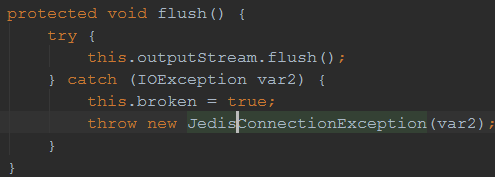


默认缓存大小是8192字节

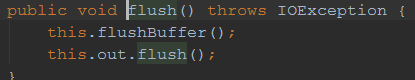
使用get命令的时候，是先写到缓存中，RedisOutputStream的byte[] buf中，数组满了后直接flush

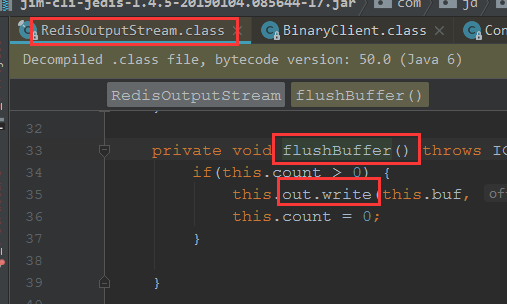




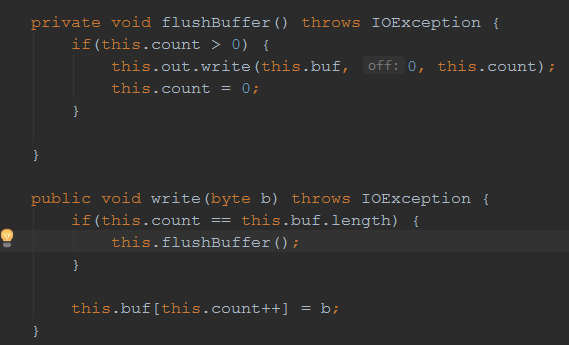


在redisOutputStream中，this.out实际是socketOutputStream

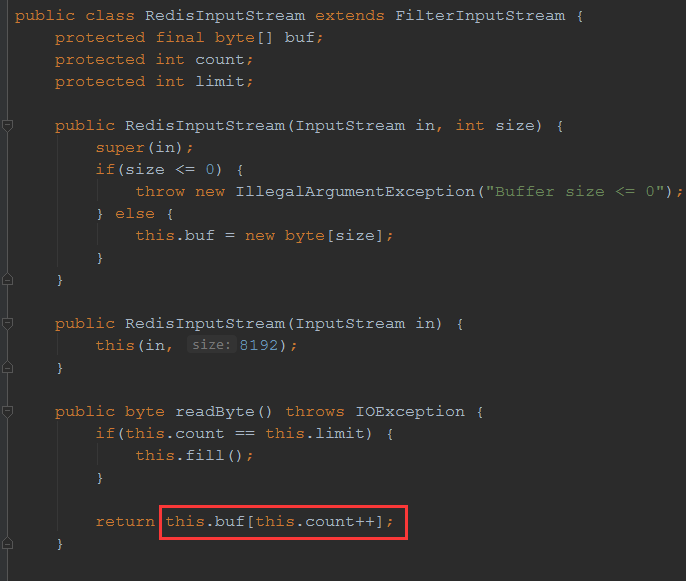




题外话，write也会调用flushBuffer



redisinputStream也是会从缓存中读取



connect时候会初始化

使用了装饰模式RedisOutputStream继承了filterOutputStream。通过构造函数将SocketOutputStream初始化进去，作为RedisOutputStream的实例变量

