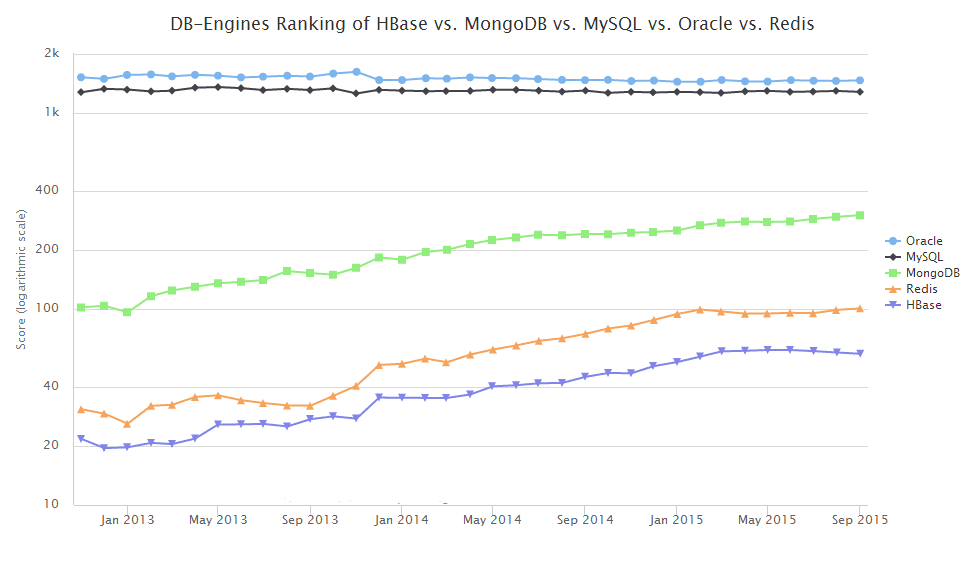
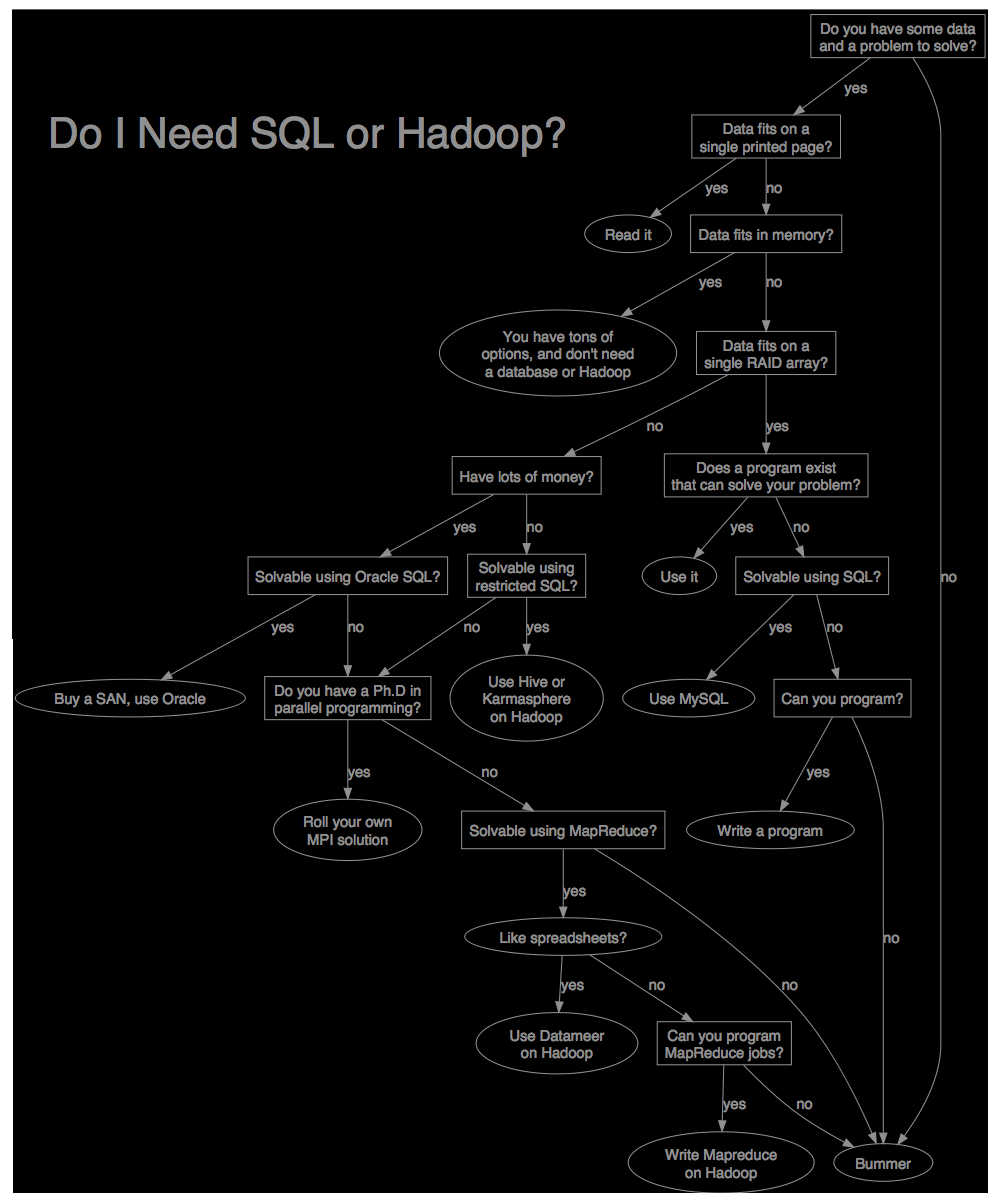
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| HBase vs. MongoDB vs. MySQL vs. Oracle vs. Redis，三大主流开源 NoSQL 数据库的 PK 两大主流传统 SQL 数据库 | | | | | | |
| **类别** | **HBase** | **MongoDB** | **MySQL** | **Oracle** | **Redis** |
| **描述** | 基于 Apache Hadoop 并提供 BigTable 能力的列存储 | 最受欢迎的文档存储数据库之一 | 广泛使用的开源 RDBMS | 广泛使用的 RDBMS | 内存中的数据结构存储，被用作数据库，缓存以及消息中间件 |
| **普及度等级(参考下文的趋势图)** | 15 级(59.03 分) | 4 级(300.57 分) | 2 级(1277.75 分) | 1 级(1463.37 分) | 10 级(100.65 分) |
| **数据库模型** | 列存储 | 文档存储 | 关系数据库系统 | 关系数据库系统 | 键-值存储 |
| **官网** | [hbase.apache.org](http://hbase.apache.org/) | [www.mongodb.org](http://www.mongodb.org/) | [www.mysql.com](http://www.mysql.com/) | [www.oracle.com/-us/-products/-database](http://www.oracle.com/us/products/database/) | [redis.io](http://redis.io/) |
| **技术文档** | [hbase.apache.org](http://hbase.apache.org/) | [docs.mongodb.org/-manual](http://docs.mongodb.org/manual/) | [dev.mysql.com/-doc](http://dev.mysql.com/doc/) | [www.oracle.com/-technetwork/-indexes/-documentation/-index.html](http://www.oracle.com/technetwork/indexes/documentation/index.html) | [redis.io/-documentation](http://redis.io/documentation) |
| **开发者** | Apache 软件基金会 | MongoDB 公司 | Oracle | Oracle | Salvatore Sanfilippo(Redis 之父) |
| **首次发行** | 2008 | 2009 | 1995 | 1980 | 2009 |
| **当前版本** | 1.1.0.1，2015 年 5 月 | 3.0.5，2015 年 7 月 | 5.6.26，2015 年 7 月 | 12 Release 1 (12.1.0.2)，2014 年 7 月 | 3.0.3，2015 年 6 月 |
| **许可** | 开源 | 开源 | 开源 | 商业 | 开源 |
| **是否数据库即服务(DBaaS)** | 否 | 否 | 否 | 否 | 否 |
| **实现语言** | Java | C++ | C 和 C++ | C 和 C++ | C |
| **服务器操作系统** | Linux Unix Windows | Linux OS X Solaris Windows | FreeBSD Linux OS X Solaris Windows | AIX HP-UX Linux OS X Solaris Windows z/OS | BSD Linux OS X Windows |
| **是否结构化数据** | 自由 | 自由 | 是 | 是 | 自由 |
| **是否预定义数据类型** | 否 | 是 | 是 | 是 | 部分 |
| **是否支持 XML** | 否 |  | 是 | 是 | 否 |
| **是否支持二级索引** | 否 | 是 | 是 | 是 | 否 |
| **是否支持 SQL** | 否 | 否 | 是 | 是 | 否 |
| **API 以及其他访问方式** | Java API RESTful HTTP API Thrift | 使用 JSON 的专用协议 | ADO.NET JDBC ODBC | ODP.NET Oracle Call Interface (OCI) JDBC ODBC | 专用协议 |
| **支持的编程语言** | C C# C++ Groovy Java PHP Python Scala | Actionscript  C C# C++ Clojure  ColdFusion  D  Dart  Delphi  Erlang Go  Groovy  Haskell Java JavaScript Lisp  Lua  MatLab  Perl PHP PowerShell  Prolog  Python R  Ruby Scala Smalltalk | Ada C C# C++ D Eiffel Erlang Haskell Java Objective-C OCaml Perl PHP Python Ruby Scheme Tcl | C C# C++ Clojure Cobol Eiffel Erlang Fortran Groovy Haskell Java JavaScript Lisp Objective C OCaml Perl PHP Python R Ruby Scala Tcl Visual Basic | C C# C++ Clojure Crystal D Dart Elixir Erlang Fancy Go Haskell Haxe Java JavaScript (Node.js) Lisp Lua MatLab Objective-C OCaml  Perl PHP Prolog Pure Data Python R Rebol Ruby Rust Scala Scheme Smalltalk Tcl |
| **是否支持服务端脚本** | 是 | JavaScript | 是 | PL/SQL | Lua |
| **是否支持触发器** | 是 | 否 | 是 | 是 | 否 |
| **切分方式** | 分片 | 分片 | 水平切分，使用 MySQL Cluster 或者 MySQL Cluster 进行分片 | 水平切分 | 分片 |
| **主从复制方式** | 多种主从复制机制 | 主-从复制 | 主-主复制 主-从复制 | 主-主复制 主-从复制 | 主-从复制 |
| **是否支持 MapReduce** | 是 | 是 | 否 | 否 | 否 |
| **分布式场景下数据一致性方法** | 立即一致 | 最终一致 立即一致 |  | 立即一致 | 最终一致 |
| **是否支持外键** | 否 | 否 | 是 | 是 | 否 |
| **是否支持事务** | 否 | 否 | ACID | ACID | 乐观锁机制，原子性执行的命令块和脚本 |
| **是否支持并发** | 是 | 是 | 是 | 是 | 是 |
| **是否支持持久化存储** | 是 | 是 | 是 | 是 | 是 |
| **是否支持内存存储** | 否 |  | 是 | 是 | 是 |
| **访问控制** | 访问控制列表(ACL) | 基于用户和角色的访问权限 | 细粒度的用户访问权限 | 根据 SQL 标准细粒度的访问权限 | 简单的基于密码的访问控制 |

**附录一：普及度等级趋势图**

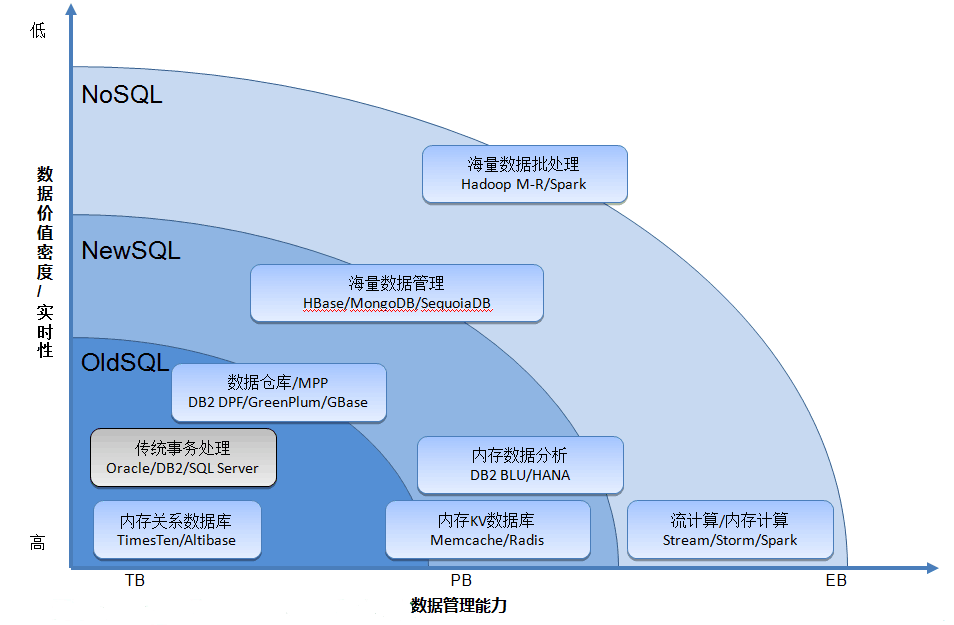


**附录二：两张图告诉你如何在 SQL、NewSQL、NoSQL 之间进行取舍**

图一：Do I Need SQL or Hadoop?



图二：SQL vs. NewSQL vs. NoSQL



**附录三：Redis 之父 Salvatore Sanfilippo 访谈节选**

Redis 是一个开源的、高级键值数据库和数据结构服务程序，其中键可以保存字符串、散列、列表、集合和有序集合。

Redis 的内核是用标准 ANSI C 写成的，基于一种事件模型。

非阻塞复制是 Redis 的设计目标。

Redis 中的复制是异步的。

目前已经为许多语言（包括大部分常用语言）提供了 Redis 库。

C client 是唯一官方支持的包装。

Redis Pub/Sub 对实时应用非常适合。

Redis 用户有的将它用作数据库，有的用作消息总线，也有的用来做 cache。

**附录四：NoSQL 数据库的类型一览表**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| NoSQL 数据库的类型一览表 | | | | | | |
| **数据库类型** | **描述** | **主流产品** | **有谁在用** | **适用场景** | **不适用场景** |
| **键值（Key-Value）数据库** | 键值数据库就像在传统语言中使用的哈希表。你可以通过 key 来添加、查询或者删除数据，鉴于使用主键访问，所以会获得不错的性能及扩展性。 | Riak、Redis、Memcached、Amazon’s Dynamo、Project Voldemort | GitHub （Riak）、BestBuy （Riak）、Twitter （Redis和Memcached）、StackOverFlow （Redis）、 Instagram （Redis）、Youtube （Memcached）、Wikipedia（Memcached） | 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。 | 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。 |
| **面向文档（Document-Oriented）数据库** | 面向文档数据库会将数据以文档的形式储存。每个文档都是自包含的数据单元，是一系列数据项的集合。每个数据项都有一个名称与对应的值，值既可以是简单的数据类型，如字符串、数字和日期等；也可以是复杂的类型，如有序列表和关联对象。数据存储的最小单位是文档，同一个表中存储的文档属性可以是不同的，数据可以使用 XML、JSON 或者 JSONB 等多种形式存储。 | MongoDB、CouchDB、RavenDB | SAP （MongoDB）、Codecademy （MongoDB）、Foursquare （MongoDB）、NBC News （RavenDB） | 1. 日志。企业环境下，每个应用程序都有不同的日志信息。Document-Oriented 数据库并没有固定的模式，所以我们可以使用它储存不同的信息。 2. 分析。鉴于它的弱模式结构，不改变模式下就可以储存不同的度量方法及添加新的度量。 | 在不同的文档上添加事务。Document-Oriented 数据库并不支持文档间的事务，如果对这方面有需求则不应该选用这个解决方案。 |
| **列存储（Wide Column Store/Column-Family）数据库** | 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。 | Cassandra、HBase | Ebay （Cassandra）、Instagram （Cassandra）、NASA （Cassandra）、Twitter （Cassandra and HBase）、Facebook （HBase）、Yahoo!（HBase） | 1. 日志。因为我们可以将数据储存在不同的列中，每个应用程序可以将信息写入自己的列族中。 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。 | 1. 如果我们需要 ACID 事务。Vassandra 就不支持事务。 2. 原型设计。如果我们分析Cassandra 的数据结构，我们就会发现结构是基于我们期望的数据查询方式而定。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。 |
| **图（Graph-Oriented）数据库** | 图数据库允许我们将数据以图的方式储存。实体会被作为顶点，而实体之间的关系则会被作为边。比如我们有三个实体，Steve Jobs、Apple 和 Next，则会有两个“Founded by”的边将 Apple 和 Next 连接到 Steve Jobs。 | Neo4J、Infinite Graph、OrientDB | Adobe （Neo4J）、Cisco （Neo4J）、T-Mobile （Neo4J） | 1. 在一些关系性强的数据中 2. 推荐引擎。如果我们将数据以图的形式表现，那么将会非常有益于推荐的制定 | 不适合的数据模型。图数据库的适用范围很小，因为很少有操作涉及到整个图。 |

**参考资料**

* [HBase vs. MongoDB vs. MySQL vs. Oracle vs. Redis Popularity](http://db-engines.com/en/system/HBase%3BMongoDB%3BMySQL%3BOracle%3BRedis)
* [一网打尽当下NoSQL类型、适用场景及使用公司](http://www.csdn.net/article/2013-07-24/2816330-how-to-choose-nosql-db)
* [Redis之父Salvatore Sanfilippo访谈](http://www.csdn.net/article/2011-01-18/290347)
* [一张图告诉你是需要 SQL 还是 Hadoop](http://blog.csdn.net/defonds/article/details/47005827)

-----------------------------------redis-------------------------------------------------

# Redis

## 介绍:

Redis:REmote DIctionary Server(远程字典服务器)

是完全开源免费的，用C语言编写的，遵守BSD协议，是一个高性能的(key/value)分布式内存数据库，基于内存运行并支持持久化的NoSQL数据库.

## 特点:

Redis支持数据的持久化，可以将内存中的数据保持在磁盘中，重启的时候可以再次加载进行使用

Redis不仅仅支持简单的key-value类型的数据，同时还提供list，set，zset，hash等数据结构的存储

Redis支持数据的备份，即master-slave模式的数据备份

## 能干嘛:

内存存储和持久化：redis支持异步将内存中的数据写到硬盘上，同时不影响继续服务

取最新N个数据的操作，如：可以将最新的10条评论的ID放在Redis的List集合里面

模拟类似于HttpSession这种需要设定过期时间的功能

发布、订阅消息系统、定时器、计数器

## 怎么玩:

持久化和复制，RDB/AOF

事务的控制

复制

## 准备工作:

在虚拟机中安装gcc

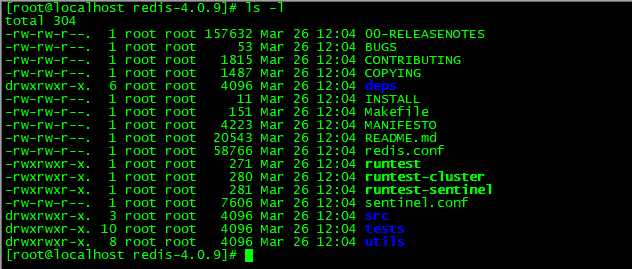
yum install gcc-c++

将redis-4.0.9.tar.gz放到虚拟机中的opt目录下,每个虚拟机下都放一个

解压tar包

tar –xzvf redis-4.0.9.tar.gz

redis的目录结构



安装完gcc后进行安装

1.make distclean

2.make

3.make install

新建存放redis.conf文件的目录

mkdir myredis

cp redis.conf /myredis/

将redis.conf修改为后台运行

# daemonize no 默认情况下，redis不是在后台运行的，如果需要在后台运行，把该项的值更改为yes

daemonize yes

查看后台redis是否启动?

ps –ef|grep redis

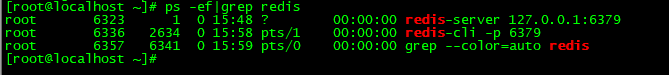
如何启动redis?

安装完redis后的安装文件默认存放在/usr/local/bin/下面;

指定哪个配置文件来启动redis->redis-server /myredis/redis.conf

指定端口启动redis->redis-cli –p 6379

启动后再查redis后台服务



redis的默认端口是多少?

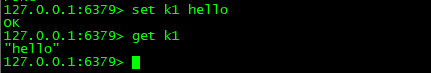
6379



进入redis



初涉redis

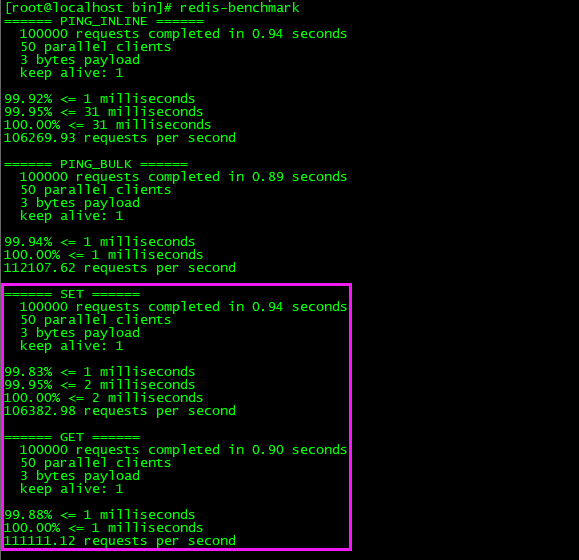


关闭redis?并查看后端进程





启动redis后,执行一下benchemark!



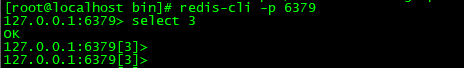
redis默认有几个库?

16



如何选择数据库?

select [index]



dbsize是什么鬼?

dbsize查看当前数据库的key的数量!

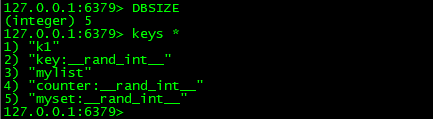


# redis五大数据类型?

string(字符串),hash(类似java里的map),list(列表),set(集合),zset(sorted set:有序集合)

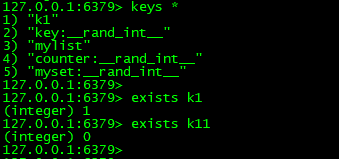
## Redis键(key)

keys\*



exists key

判断某个key是否存在

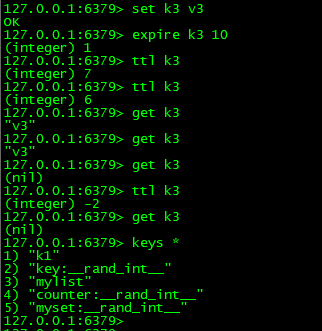


move key db

当前库就没有了,被移除了

expire key

为给定的key设置过期时间(秒钟)



给给定的值设置过期时间,过期时间后将无法访问!

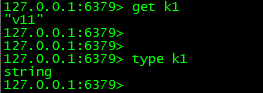
ttl key

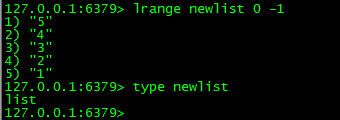
查看还有多少秒过期,-1永不过期,-2表示已过期



type key

查看你的key是什么类型

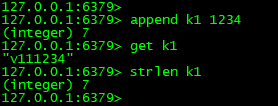
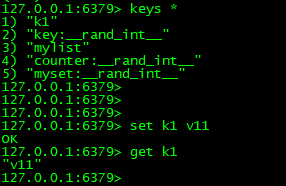




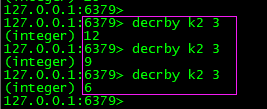
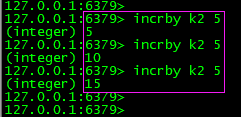
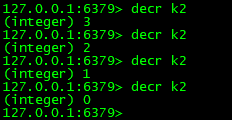
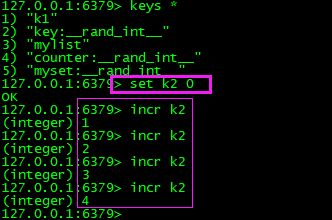
## Redis字符串(String)

单值单value

set/get/del/append/strlen

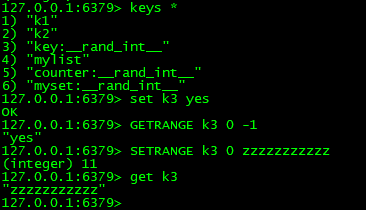


incr/decr/incrby/decrby,一定要是数字才能进行加减

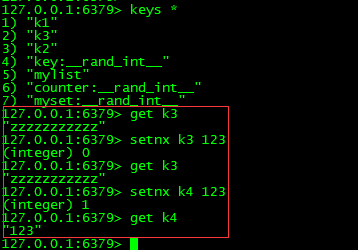
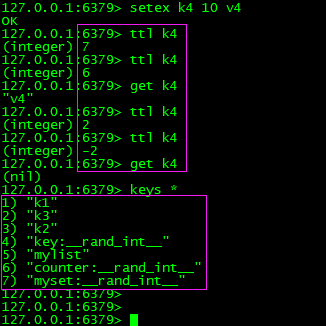


getrange/setrange

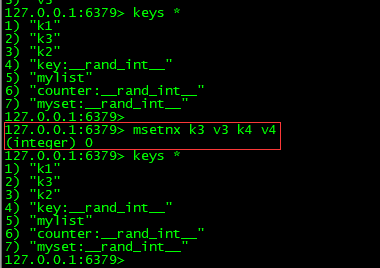
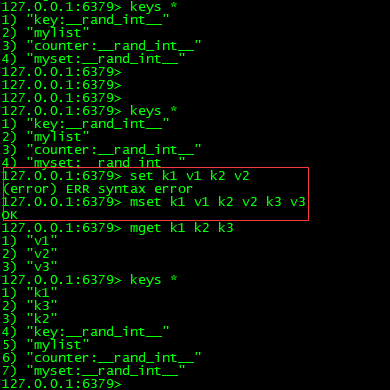
getrange:获取指定区间范围内的值，类似between......and的关系从零到负一表示全部



setex(set with expire)键秒值/setnx(set if not exist)



mset/mget/msetnx

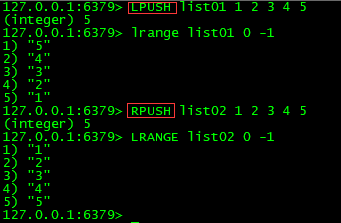


getset(先get再set)

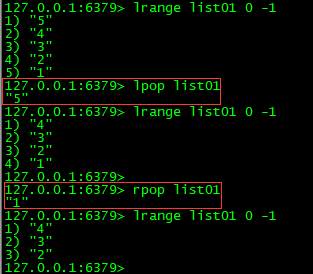
## Redis列表(List)

常用,单值多value

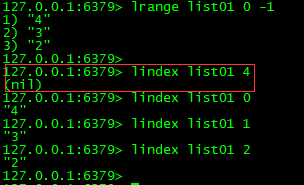
lpush/rpush/lrange



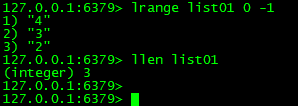
lpop/rpop



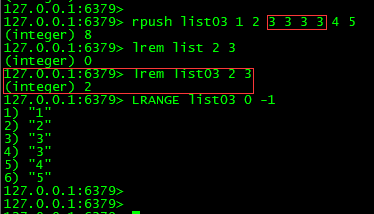
lindex,按照索引下标获得元素(从上到下)



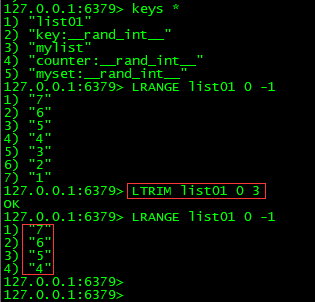
llen



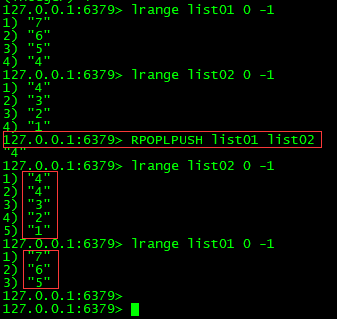
lrem key删N个value



ltrim key开始index结束index,截取指定范围的值后再赋值给key



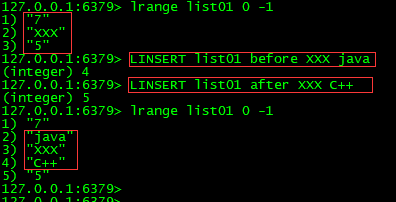
rpoplpush [源列表] [目的列表]



lset key index value



linsert key before/after 值1 值2



性能总结

它是一个字符串链表，left、right都可以插入添加；

如果键不存在，创建新的链表；

如果键已存在，新增内容；

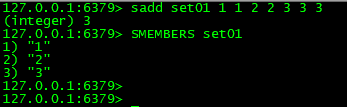
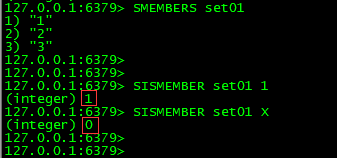
如果值全移除，对应的键也就消失了。

链表的操作无论是头和尾效率都极高，但假如是对中间元素进行操作，效率就很惨淡了。

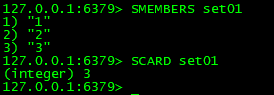
## Redis集合(set)

常用,单值多value

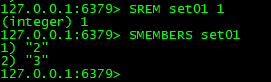
sadd/smemebers/sismemeber

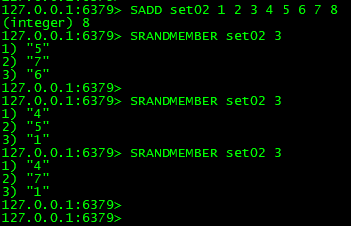
scard,获取集合里面的元素个数



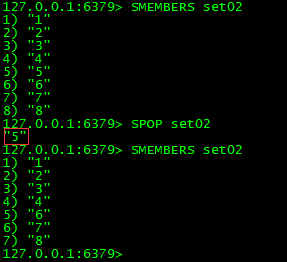
srem key value删除集合中元素



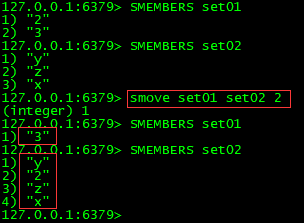
srandmemeber key某个整数(随机出几个数)



spop key随机出栈



smove key1 key2 在key1里某个值,作用是将key1里面的某个值赋值给key2

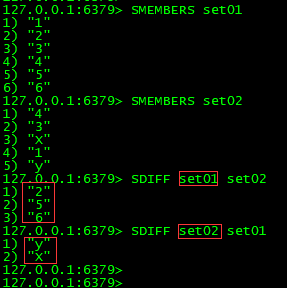
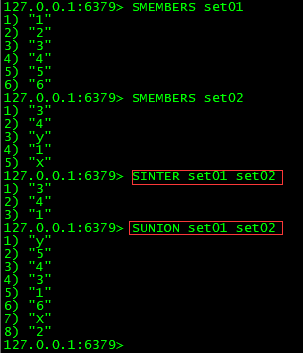


数据集合类

差集:sdiff

交集:sinter

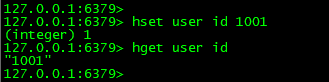
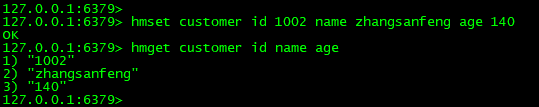
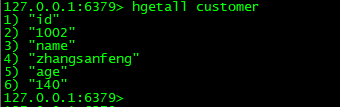
并集:sunion

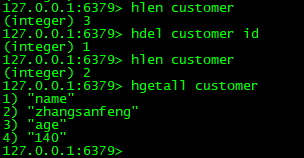
## Redis哈希(hash)

常用,KV模式不变,但V是一个键值对

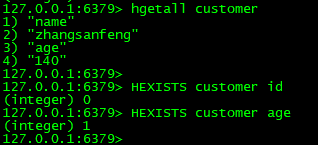
hset/hget/hmset/hmget/hgetall/hdel

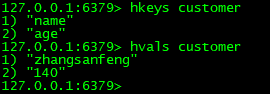
hlen



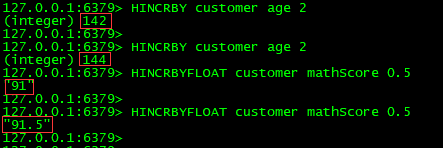
hexists key 在key里面的某个值的key



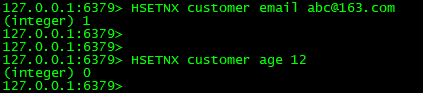
hkeys/hvals



hincrby/hincrbyfloat

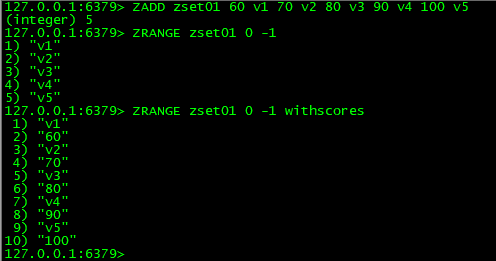


hsetnx



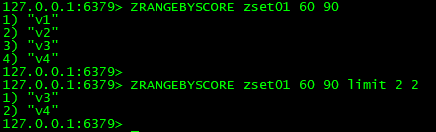
## Redis有序集合Zset(sorted set)

zadd/zrange

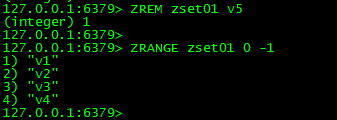


zrangebyscore key开始score结束score

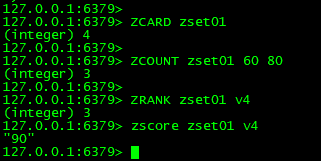




zrem key是某score下对应的value值,作用是删除元素



zcard/zcount key score区间/zrank key values值,作用是获得下标值/zscore key对应值,获得分数



zrevrank key values值,作用是逆序获得下标

zrevrange



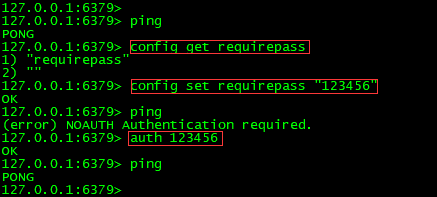
zrevrangebyscore key 结束score开始score



# Redis.conf配置文件详细讲解

#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

## 给redis设置一个auth密码



# Redis内存持久化

RDB（Redis DataBase）

AOF（Append Only File）

## RDB

RDB是什么?

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。

Fork是什么?

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

RDB保存的是什么?

rdb 保存的是dump.rdb文件

redis中aof和rdb可不可以共存?

可以共存



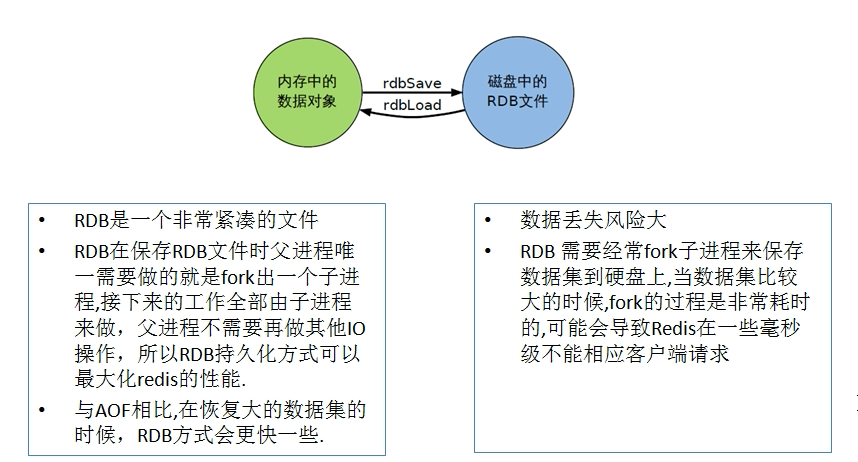
redis中aof和rdb都存在的情况下,先加载哪个?

先加载aof,如果aof文件损坏,则redis服务无法启动!

如果aof文件损坏,则如何进行修复?

redis-check-aof --fix appendonly.aof

小总结



## AOF

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

rewrite是什么：

AOF采用文件追加方式，文件会越来越大为避免出现此种情况，新增了重写机制,当AOF文件的大小超过所设定的阈值时，Redis就会启动AOF文件的内容压缩，只保留可以恢复数据的最小指令集.可以使用命令bgrewriteaof

rewrite重写原理

AOF文件持续增长而过大时，会fork出一条新进程来将文件重写(也是先写临时文件最后再rename)，遍历新进程的内存中数据，每条记录有一条的Set语句。重写aof文件的操作，并没有读取旧的aof文件，而是将整个内存中的数据库内容用命令的方式重写了一个新的aof文件，这点和快照有点类似

rewrite触发机制

Redis会记录上次重写时的AOF大小，默认配置是当AOF文件大小是上次rewrite后大小的一倍且文件大于64M时触发

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

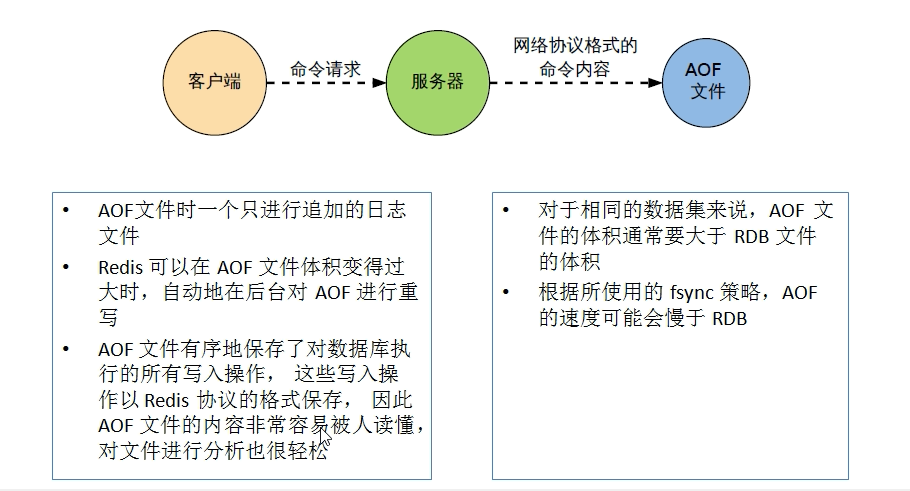
不同步：appendfsync no 从不同步

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

小总结



# Redis事务

redis事务是什么?

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

悲观锁

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

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

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

# Redis发布订阅

进程间的一种消息通信模式：发送者(pub)发送消息，订阅者(sub)接收消息。

案例

先订阅后发布后才能收到消息，

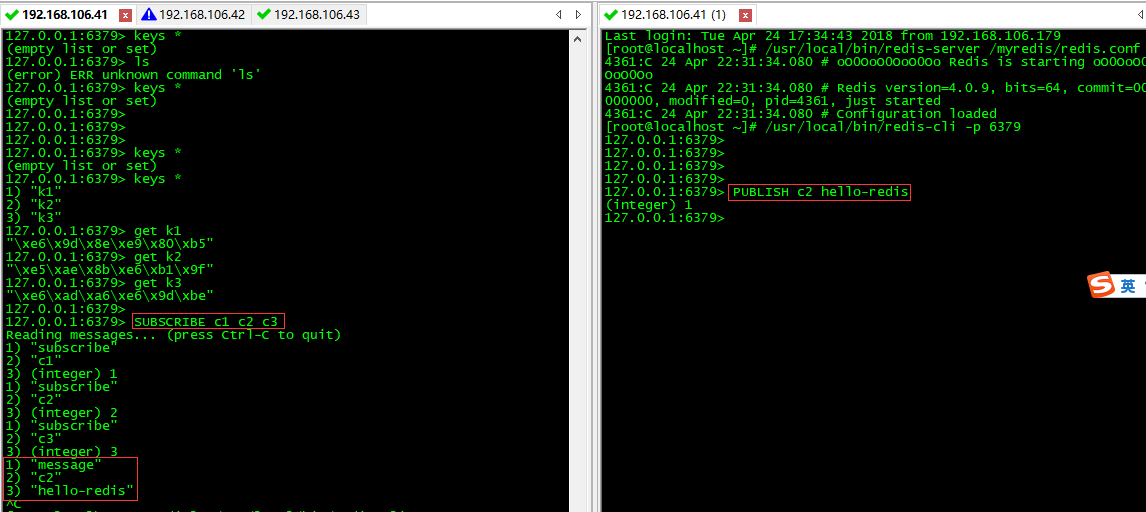
1 可以一次性订阅多个，SUBSCRIBE c1 c2 c3

2 消息发布，PUBLISH c2 hello-redis

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

3 订阅多个，通配符\*， PSUBSCRIBE new\*

4 收取消息， PUBLISH new1 redis2015



# Redis复制(Master/Slave)

是什么?

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

能干嘛?

读写分离

容灾恢复

修改配置文件细节操作

1: 拷贝多个redis.conf文件

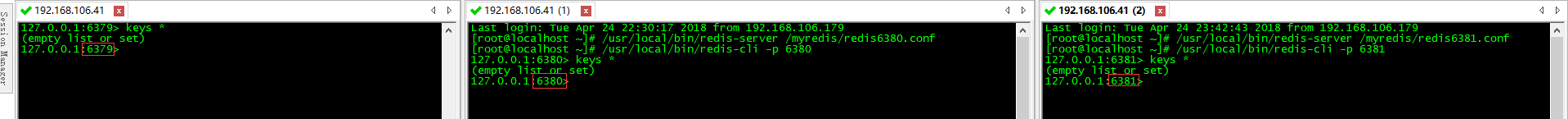
2: 开启daemonize yes

3: pid文件名字

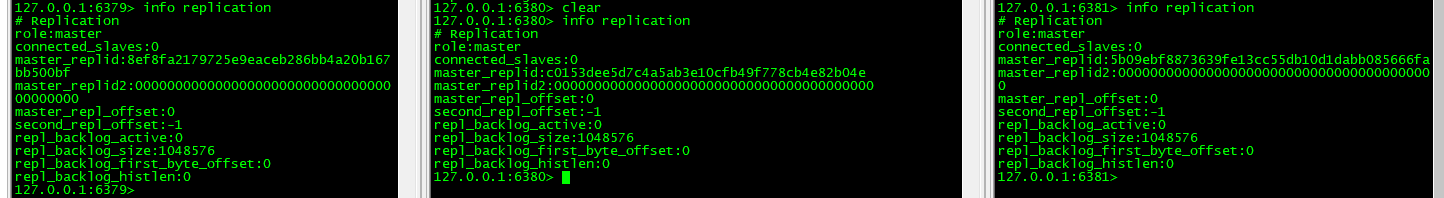
4: 指定端口

5: log文件名字

6: dump.rdb名字

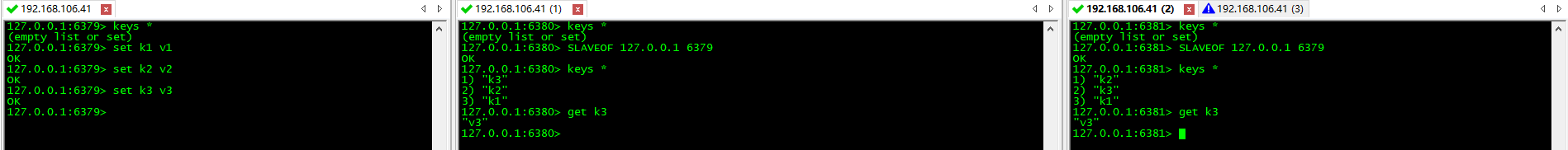


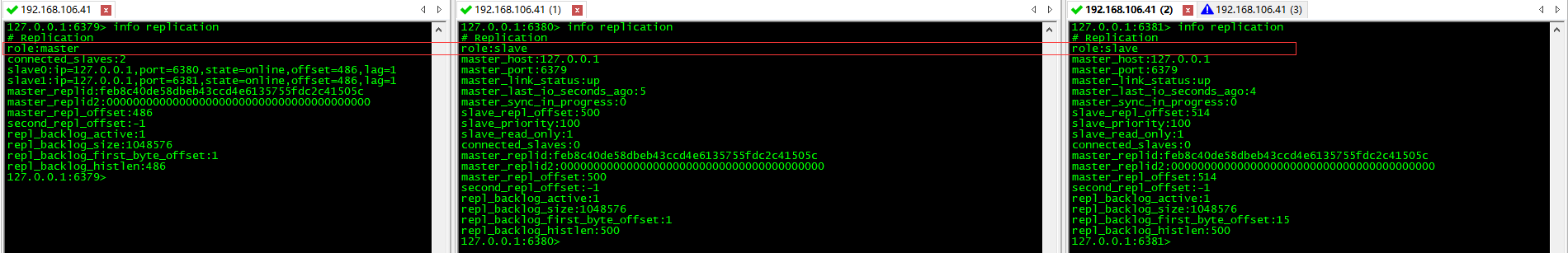
没有设置主从的时候



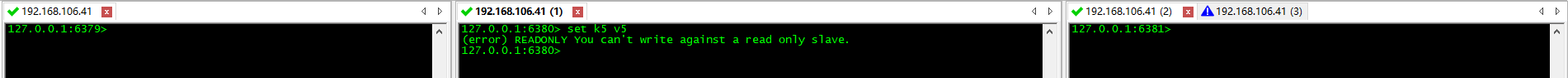
一主二仆



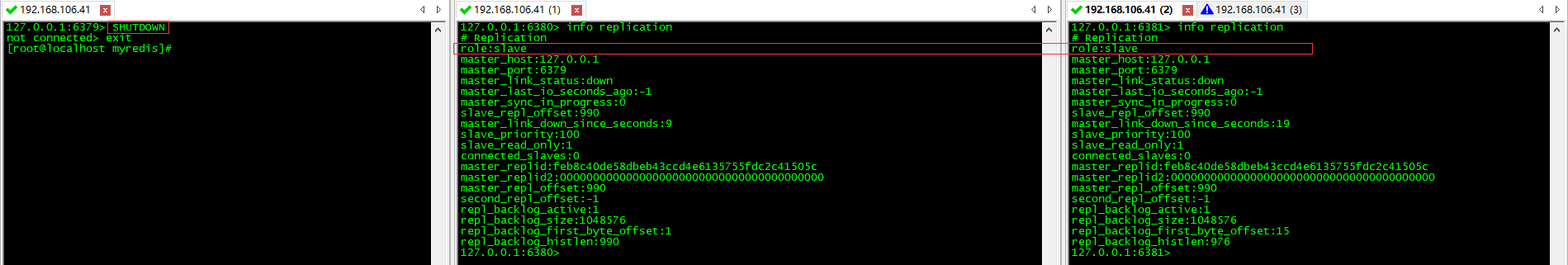




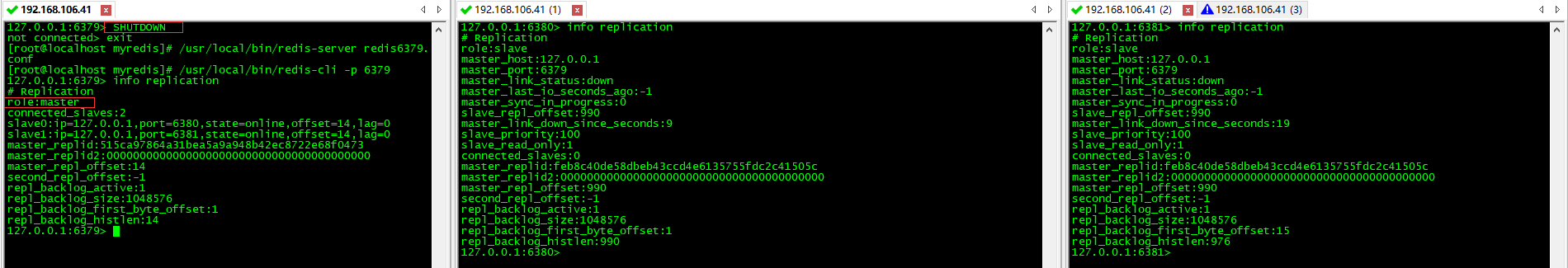
从是只读模式



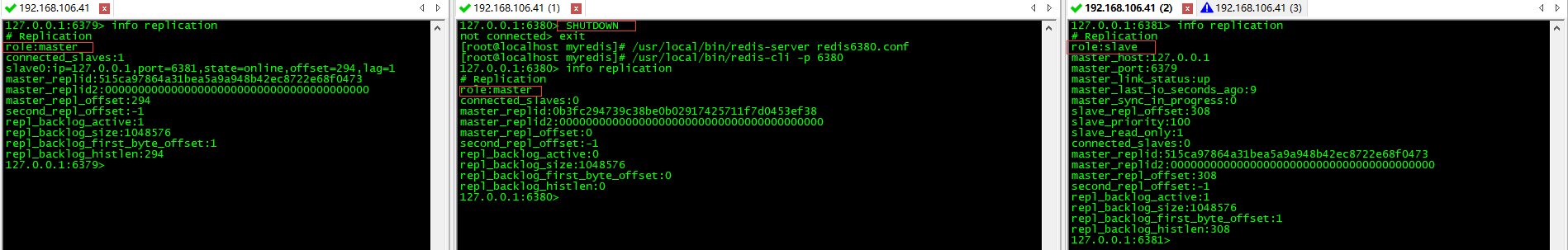
Master Down掉后



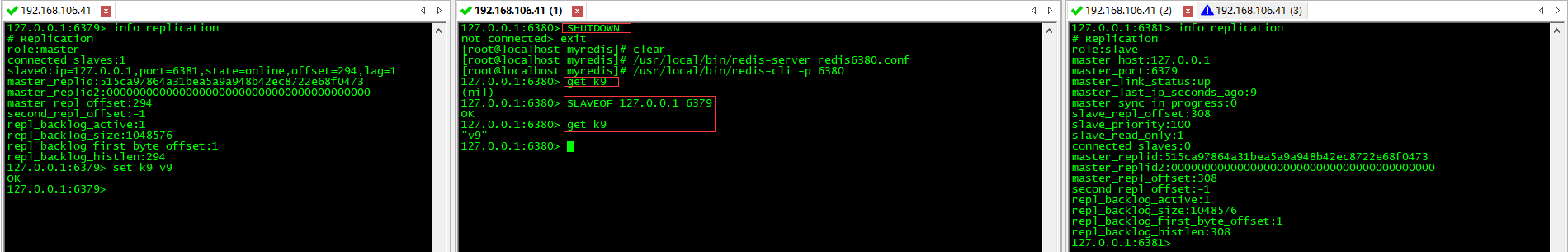
6379再次重启后依然是Master



80端口Down后再次重启,这次变为自己的Master



80端口再次启动并再次slaveof 端口的前后



# Redis哨兵模式(sentinel)

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 被监控数据库名字(自己起名字) 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 /myredis/sentinel.conf

上述目录依照各自的实际情况配置，可能目录不同

一组sentinel能同时监控多个Master

# 连接

## 如何在虚拟机中开启6379端口?

临时关闭防火墙

systemctl stop firewalld

永久防火墙开机自启动

systemctl disable firewalld

临时打开防火墙

systemctl start firewalld

防火墙开机启动

systemctl enable firewalld

查看防火墙状态

systemctl status firewalld

通过systemctl status firewalld查看firewalld状态，发现当前是dead状态，即防火墙未开启。

通过systemctl start firewalld开启防火墙，没有任何提示即开启成功。

再次通过systemctl status firewalld查看firewalld状态，显示running即已开启了。

再次执行执行firewall-cmd --permanent --zone=public --add-port=6379/tcp，提示success，表示设置成功，这样就可以继续后面的设置了。

## 本机telnet虚拟机报错?

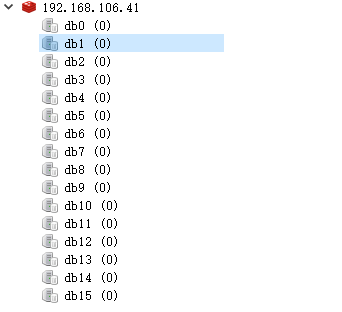
-DENIED Redis is running in protected mode because protected mode is enabled, no bind address was specified, no authentication password is requested to clients. In this mode connections are only accepted from the loopback interface.

需要修改redis.conf文件

protected-mode no

重启redis服务后即可!

## 使用RDM连接虚拟机的redis?



# Jedis

## 本机连接Redis\_Jedis测试联通?

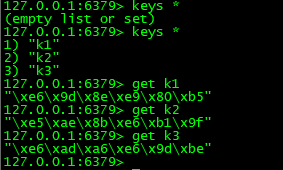
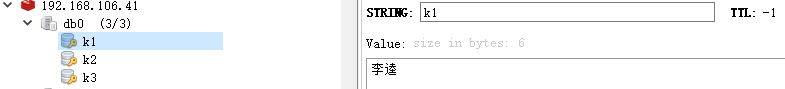
**package com.**skytech.redis;  
  
**import** redis.clients.jedis.Jedis;  
  
**public class TestPing** {  
 **public static void** main(**String**[] args) {  
 Jedis jedis = **new** Jedis("192.168.106.41", **6379**);  
 **String** str = jedis.ping();  
 **System**.***out***.println(str);  
 }  
}

测试成功结果



## Jedis的set操作?

**package com.**skytech.redis;  
  
**import** redis.clients.jedis.Jedis;  
  
**public class TestPing** {  
 **public static void** main(**String**[] args) {  
 Jedis jedis = **new** Jedis("192.168.106.41", **6379**);  
  
 jedis.set("k1", "李逵");  
 jedis.set("k2", "宋江");  
 jedis.set("k3", "武松");  
 }  
}

## 封装的redis基本命令

**package exam**;  
  
  
**public class redisCluster** {  
 **private static final** Log ***log*** = LogFactory.getLog(**redisCluster**.**class**);  
 JedisCluster *jedisCluster* = **null**;  
 ClassPathXmlApplicationContext *context* = **null**;  
  
 /\*\*  
 \* 构造函数，初始化jedis连接池  
 \*/  
 **public redisCluster**() {  
 **try** {  
 *context* = **new** ClassPathXmlApplicationContext("classpath:spring/spring-context.xml");  
 *context*.start();  
 *jedisCluster* = (JedisCluster) *context*.getBean("jedisCluster");  
 } **catch** (**Exception** e) {  
 ***log***.error("==>redisCluster context start error:", e);  
 *context*.stop();  
 **System**.exit(**0**);  
 }  
 }  
  
 /\*\*  
 \* 关闭连接  
 \*/  
 **public void** closeRedisCluster() {  
 *jedisCluster*.close();  
 }  
 //Key(键)  
  
 /\*\*  
 \* 删除指定的key  
 \*  
 \* @param key  
 \* @return 删除的数量  
 \*/  
 **public Long** delete(**String** key) {  
 **return** *jedisCluster*.del(key);  
 }  
  
 /\*\*  
 \* 检查指定的key是否存在  
 \*  
 \* @param key  
 \* @return 存在：返回1 不存在：返回0  
 \*/  
 **public Boolean** exists(**String** key) {  
 **return** *jedisCluster*.exists(key);  
 }  
  
 /\*\*  
 \* 为给定 key 设置生存时间，当 key 过期时(生存时间为 0 )，它会被自动删除  
 \* 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  
 \*  
 \* @param key  
 \* @param seconds 秒  
 \* @return 成功:返回1 失败：返回0  
 \*/  
 **public Long** expire(**String** key, **int** seconds) {  
 **return** *jedisCluster*.expire(key, seconds);  
 }  
  
 /\*\*  
 \* EXPIREAT 的作用和 EXPIRE 类似，都用于为 key 设置生存时间  
 \* 不同在于 EXPIREAT 命令接受的时间参数是 UNIX 时间戳(unix timestamp)  
 \*  
 \* @param key  
 \* @param unixTime 秒  
 \* @return 成功:返回1 失败：返回0  
 \*/  
 **public long** expireAt(**String** key, **long** unixTime) {  
 **return** *jedisCluster*.expireAt(key, unixTime);  
 }  
  
 /\*\*  
 \* 将当前数据库的 key 移动到给定的数据库 db 当中  
 \* 如果当前数据库(源数据库)和给定数据库(目标数据库)有相同名字的给定 key ，或者 key 不存在于当前数据库，那么 MOVE 没有任何效果  
 \*  
 \* @param key  
 \* @param dbIndex  
 \* @return 成功：返回1 失败：返回0  
 \*/  
 **public Long** move(**String** key, **int** dbIndex) {  
 **return** *jedisCluster*.move(key, dbIndex);  
 }  
  
 /\*\*  
 \* 移除给定 key 的生存时间，将这个 key 从『易失的』(带生存时间 key )转换成『持久的』(一个不带生存时间、永不过期的 key )  
 \*  
 \* @param key  
 \* @return 成功：返回1 失败：返回0  
 \*/  
 **public long** persist(**String** key) {  
 **return** *jedisCluster*.persist(key);  
 }  
  
 /\*\*  
 \* 为给定 key 设置生存时间，当 key 过期时(生存时间为 0 )，它会被自动删除  
 \* 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  
 \*  
 \* @param key  
 \* @param seconds 毫秒  
 \* @return 成功:返回1 失败：返回0  
 \*/  
 **public long** pexpire(**String** key, **long** milliseconds) {  
 **return** *jedisCluster*.pexpire(key, milliseconds);  
 }  
  
 /\*\*  
 \* EXPIREAT 的作用和 EXPIRE 类似，都用于为 key 设置生存时间  
 \* 不同在于 EXPIREAT 命令接受的时间参数是 UNIX 时间戳(unix timestamp)  
 \*  
 \* @param key  
 \* @param unixTime 毫秒  
 \* @return 成功:返回1 失败：返回0  
 \*/  
 **public long** pexpireAt(**String** key, **long** millisecondsTimestamp) {  
 **return** *jedisCluster*.pexpireAt(key, millisecondsTimestamp);  
 }  
  
 /\*\*  
 \* 返回或保存给定列表、集合、有序集合 key 中经过排序的元素  
 \* 排序默认以数字作为对象，值被解释为双精度浮点数，然后进行比较  
 \*  
 \* @param key  
 \* @return  
 \*/  
 **public** List<**String**> sort(**String** key) {  
 **return** *jedisCluster*.sort(key);  
 }  
  
 /\*\*  
 \* 以秒为单位，返回给定 key 的剩余生存时间(TTL, time to live)  
 \*  
 \* @param key  
 \* @return key不存在：返回-2 key存在但没有设置生存时间：返回-1  
 \*/  
 **public long** ttl(**String** key) {  
 **return** *jedisCluster*.ttl(key);  
 }  
  
 /\*\*  
 \* 返回 key 所储存的值的类型  
 \*  
 \* @param key  
 \* @return none (key不存在) string (字符串) list (列表) set (集合) zset (有序集) hash (哈希表)  
 \*/  
 **public String** type(**String** key) {  
 **return** *jedisCluster*.type(key);  
 }  
 //String(字符串)  
  
 /\*\*  
 \* 如果 key 已经存在并且是一个字符串， APPEND 命令将 value 追加到 key 原来的值的末尾  
 \* 如果 key 不存在， APPEND 就简单地将给定 key 设为 value ，就像执行 SET key value 一样  
 \*  
 \* @param key  
 \* @param value  
 \* @return 追加 value 之后， key 中字符串的长度  
 \*/  
 **public Long** append(**String** key, **String** value) {  
 **return** *jedisCluster*.append(key, value);  
 }  
  
 /\*\*  
 \* 计算给定字符串中，被设置为 1 的比特位的数量  
 \*  
 \* @param key  
 \* @return 被设置为 1 的位的数量  
 \*/  
 **public Long** bitCount(**String** key) {  
 **return** *jedisCluster*.bitcount(key);  
 }  
  
 /\*\*  
 \* 通过指定额外的 start 或 end 参数，可以让计数只在特定的位上进行  
 \*  
 \* @param key  
 \* @param start  
 \* @param end  
 \* @return 被设置为 1 的位的数量  
 \*/  
 **public Long** bitCount(**String** key, **Long** start, **Long** end) {  
 **return** *jedisCluster*.bitcount(key, start, end);  
 }  
  
 /\*\*  
 \* 将 key 中储存的数字值减一  
 \* 如果 key 不存在，那么 key 的值会先被初始化为 0 ，然后再执行 DECR 操作  
 \* 如果值包含错误的类型，或字符串类型的值不能表示为数字，那么返回一个错误  
 \*  
 \* @param key  
 \* @return 执行 DECR 命令之后 key 的值  
 \*/  
 **public Long** decr(**String** key) {  
 **return** *jedisCluster*.decr(key);  
 }  
  
 /\*\*  
 \* 将 key 所储存的值减去减量 decrement  
 \* 如果 key 不存在，那么 key 的值会先被初始化为 0 ，然后再执行 DECRBY 操作  
 \* 如果值包含错误的类型，或字符串类型的值不能表示为数字，那么返回一个错误  
 \*  
 \* @param key  
 \* @param integer  
 \* @return 减去 decrement 之后， key 的值  
 \*/  
 **public Long** decrBy(**String** key, **long** integer) {  
 **return** *jedisCluster*.decrBy(key, integer);  
 }  
  
 /\*\*  
 \* 返回 key 所关联的字符串值  
 \* 如果 key 不存在那么返回特殊值 nil  
 \* 假如 key 储存的值不是字符串类型，返回一个错误，因为 GET 只能用于处理字符串值  
 \*  
 \* @param key  
 \* @return 当 key 不存在时，返回 nil ，否则，返回 key 的值,如果 key 不是字符串类型，那么返回一个错误  
 \*/  
 **public String** get(**String** key) {  
 **return** *jedisCluster*.get(key);  
 }  
  
 /\*\*  
 \* 对 key 所储存的字符串值，获取指定偏移量上的位(bit)  
 \* 当 offset 比字符串值的长度大，或者 key 不存在时，返回 0  
 \*  
 \* @param key  
 \* @param offset  
 \* @return 字符串值指定偏移量上的位(bit)  
 \*/  
 **public Boolean** getbit(**String** key, **Long** offset) {  
 **return** *jedisCluster*.getbit(key, offset);  
 }  
  
 /\*\*  
 \* 返回 key 中字符串值的子字符串，字符串的截取范围由 start 和 end 两个偏移量决定(包括 start 和 end 在内)  
 \* 负数偏移量表示从字符串最后开始计数， -1 表示最后一个字符， -2 表示倒数第二个，以此类推。  
 \*  
 \* @param key  
 \* @param startOffset  
 \* @param endOffset  
 \* @return 截取得出的子字符串  
 \*/  
 **public String** getrange(**String** key, **long** startOffset, **long** endOffset) {  
 **return** *jedisCluster*.getrange(key, startOffset, endOffset);  
 }  
  
 /\*\*  
 \* 将给定 key 的值设为 value ，并返回 key 的旧值(old value)  
 \* 当 key 存在但不是字符串类型时，返回一个错误  
 \*  
 \* @param key  
 \* @param value  
 \* @return 返回给定 key 的旧值,当 key 没有旧值时，也即是， key 不存在时，返回 nil  
 \*/  
 **public String** getSet(**String** key, **String** value) {  
 **return** *jedisCluster*.getSet(key, value);  
 }  
  
 /\*\*  
 \* 将 key 中储存的数字值增一  
 \* 如果 key 不存在，那么 key 的值会先被初始化为 0 ，然后再执行 INCR 操作  
 \* 如果值包含错误的类型，或字符串类型的值不能表示为数字，那么返回一个错误  
 \*  
 \* @param key  
 \* @return 执行 INCR 命令之后 key 的值  
 \*/  
 **public Long** incr(**String** key) {  
 **return** *jedisCluster*.incr(key);  
 }  
  
 /\*\*  
 \* 将 key 所储存的值加上增量 increment  
 \* 如果 key 不存在，那么 key 的值会先被初始化为 0 ，然后再执行 INCRBY 命令  
 \* 如果值包含错误的类型，或字符串类型的值不能表示为数字，那么返回一个错误  
 \*  
 \* @param key  
 \* @param integer  
 \* @return 加上 increment 之后， key 的值  
 \*/  
 **public Long** incrBy(**String** key, **long** integer) {  
 **return** *jedisCluster*.incrBy(key, integer);  
 }  
  
 /\*\*  
 \* 为 key 中所储存的值加上浮点数增量 increment  
 \* 如果 key 不存在，那么 INCRBYFLOAT 会先将 key 的值设为 0 ，再执行加法操作  
 \* 如果命令执行成功，那么 key 的值会被更新为（执行加法之后的）新值，并且新值会以字符串的形式返回给调用者  
 \*  
 \* @param key  
 \* @param value  
 \* @return 执行命令之后 key 的值  
 \*/  
 **public double** incrByFloat(**String** key, **double** value) {  
 **return** *jedisCluster*.incrByFloat(key, value);  
 }  
  
 /\*\*  
 \* 将字符串值 value 关联到 key  
 \* 如果 key 已经持有其他值， SET 就覆写旧值，无视类型  
 \*  
 \* @param key  
 \* @param value  
 \* @return 返回ok  
 \*/  
 **public String** set(**String** key, **String** value) {  
 **return** *jedisCluster*.set(key, value);  
 }  
  
 /\*\*  
 \* 对 key 所储存的字符串值，设置或清除指定偏移量上的位(bit)  
 \* 位的设置或清除取决于 value 参数，可以是 0 也可以是 1  
 \* 当 key 不存在时，自动生成一个新的字符串值  
 \*  
 \* @param key  
 \* @param offset  
 \* @param value  
 \* @return 指定偏移量原来储存的位  
 \*/  
 **public Boolean** setbit(**String** key, **Long** offset, **String** value) {  
 **return** *jedisCluster*.setbit(key, offset, value);  
 }  
  
 /\*\*  
 \* 对 key 所储存的字符串值，设置或清除指定偏移量上的位(bit)  
 \* 位的设置或清除取决于 value 参数，可以是 0 也可以是 1  
 \* 当 key 不存在时，自动生成一个新的字符串值  
 \*  
 \* @param key  
 \* @param offset  
 \* @param value  
 \* @return 指定偏移量原来储存的位  
 \*/  
 **public Boolean** setbit(**String** key, **long** offset, **Boolean** value) {  
 **return** *jedisCluster*.setbit(key, offset, value);  
 }  
  
 /\*\*  
 \* 将值 value 关联到 key ，并将 key 的生存时间设为 seconds (以秒为单位)  
 \* 如果 key 已经存在， SETEX 命令将覆写旧值  
 \*  
 \* @param key  
 \* @param seconds  
 \* @param value  
 \* @return 设置成功时返回 OK ,当 seconds 参数不合法时，返回一个错误  
 \*/  
 **public String** setex(**String** key, **int** seconds, **String** value) {  
 **return** *jedisCluster*.setex(key, seconds, value);  
 }  
  
 /\*\*  
 \* 将 key 的值设为 value ，当且仅当 key 不存在  
 \* 若给定的 key 已经存在，则 SETNX 不做任何动作  
 \*  
 \* @param key  
 \* @param value  
 \* @return 设置成功，返回 1,设置失败，返回 0  
 \*/  
 **public Long** setnx(**String** key, **String** value) {  
 **return** *jedisCluster*.setnx(key, value);  
 }  
  
 /\*\*  
 \* 用 value 参数覆写(overwrite)给定 key 所储存的字符串值，从偏移量 offset 开始  
 \* 不存在的 key 当作空白字符串处理  
 \*  
 \* @param key  
 \* @param offset  
 \* @param value  
 \* @return 被 SETRANGE 修改之后，字符串的长度  
 \*/  
 **public Long** setrange(**String** key, **Long** offset, **String** value) {  
 **return** *jedisCluster*.setrange(key, offset, value);  
 }  
  
 /\*\*  
 \* 返回 key 所储存的字符串值的长度  
 \* 当 key 储存的不是字符串值时，返回一个错误  
 \*  
 \* @param key  
 \* @return 字符串值的长度, 当 key 不存在时，返回 0  
 \*/  
 **public Long** strlen(**String** key) {  
 **return** *jedisCluster*.strlen(key);  
 }  
 //Hash(哈希表)  
  
 /\*\*  
 \* 删除哈希表 key 中的一个或多个指定域，不存在的域将被忽略  
 \*  
 \* @param key  
 \* @param fields 格式为：field,field,field....  
 \* @return 被成功移除的域的数量，不包括被忽略的域  
 \*/  
 **public Long** hdel(**String** key, **String** fields) {  
 **return** *jedisCluster*.hdel(key, fields);  
 }  
  
 /\*\*  
 \* 查看哈希表 key 中，给定域 field 是否存在  
 \*  
 \* @param key  
 \* @param field  
 \* @return 如果哈希表含有给定域，返回 1,如果哈希表不含有给定域，或 key 不存在，返回 0  
 \*/  
 **public Boolean** hexists(**String** key, **String** field) {  
 **return** *jedisCluster*.hexists(key, field);  
 }  
  
 /\*\*  
 \* 返回哈希表 key 中给定域 field 的值  
 \*  
 \* @param key  
 \* @param field  
 \* @return 给定域的值, 当给定域不存在或是给定 key 不存在时，返回 nil  
 \*/  
 **public String** hget(**String** key, **String** field) {  
 **return** *jedisCluster*.hget(key, field);  
 }  
  
 /\*\*  
 \* 返回哈希表 key 中，所有的域和值  
 \* 在返回值里，紧跟每个域名(field name)之后是域的值(value)，所以返回值的长度是哈希表大小的两倍  
 \*  
 \* @param key  
 \* @return 以列表形式返回哈希表的域和域的值, 若 key 不存在，返回空列表  
 \*/  
 **public** Map<**String**, **String**> hgetAll(**String** key) {  
 **return** *jedisCluster*.hgetAll(key);  
 }  
  
 /\*\*  
 \* 为哈希表 key 中的域 field 的值加上增量 increment  
 \* 增量也可以为负数，相当于对给定域进行减法操作  
 \* 如果 key 不存在，一个新的哈希表被创建并执行 HINCRBY 命令  
 \* 如果域 field 不存在，那么在执行命令前，域的值被初始化为 0  
 \* 对一个储存字符串值的域 field 执行 HINCRBY 命令将造成一个错误  
 \*  
 \* @param key  
 \* @param field  
 \* @param value  
 \* @return 执行 HINCRBY 命令之后，哈希表 key 中域 field 的值  
 \*/  
 **public Long** hincrBy(**String** key, **String** field, **long** value) {  
 **return** *jedisCluster*.hincrBy(key, field, value);  
 }  
  
 /\*\*  
 \* 返回哈希表 key 中的所有域  
 \*  
 \* @param key  
 \* @return 一个包含哈希表中所有域的表, 当 key 不存在时，返回一个空表  
 \*/  
 **public** Set<**String**> hkeys(**String** key) {  
 **return** *jedisCluster*.hkeys(key);  
 }  
  
 /\*\*  
 \* 返回哈希表 key 中域的数量  
 \*  
 \* @param key  
 \* @return 哈希表中域的数量, 当 key 不存在时，返回 0  
 \*/  
 **public Long** hlen(**String** key) {  
 **return** *jedisCluster*.hlen(key);  
 }  
  
 /\*\*  
 \* 返回哈希表 key 中，一个或多个给定域的值  
 \* 如果给定的域不存在于哈希表，那么返回一个 nil 值  
 \* 因为不存在的 key 被当作一个空哈希表来处理，所以对一个不存在的 key 进行 HMGET 操作将返回一个只带有 nil 值的表  
 \*  
 \* @param key  
 \* @param fields 格式为：field,field,field........  
 \* @return 一个包含多个给定域的关联值的表，表值的排列顺序和给定域参数的请求顺序一样  
 \*/  
 **public** List<**String**> hmget(**String** key, **String** fields) {  
 **return** *jedisCluster*.hmget(key, fields);  
 }  
  
 /\*\*  
 \* 同时将多个 field-value (域-值)对设置到哈希表 key 中  
 \* 此命令会覆盖哈希表中已存在的域  
 \* 如果 key 不存在，一个空哈希表被创建并执行 HMSET 操作  
 \*  
 \* @param key  
 \* @param fieldValues field,value,field,value....  
 \* @return 如果命令执行成功，返回 OK ,当 key 不是哈希表(hash)类型时，返回一个错误  
 \*/  
 **public String** hmset(**String** key, **String** fieldValues) {  
 **String**[] fieldValue = fieldValues.split(",");  
 Map<**String**, **String**> map = **new** HashMap<**String**, **String**>();  
 **for** (**int** i = **0**; i < fieldValue.*length*; i++) {  
 map.put(fieldValue[i], fieldValue[i + **1**]);  
 }  
 **return** *jedisCluster*.hmset(key, map);  
 }  
  
 /\*\*  
 \* 将哈希表 key 中的域 field 的值设为 value  
 \* 如果 key 不存在，一个新的哈希表被创建并进行 HSET 操作  
 \* 如果域 field 已经存在于哈希表中，旧值将被覆盖  
 \*  
 \* @param key  
 \* @param field  
 \* @param value  
 \* @return 如果 field 是哈希表中的一个新建域，并且值设置成功，返回 1,如果哈希表中域 field 已经存在且旧值已被新值覆盖，返回 0  
 \*/  
 **public Long** hset(**String** key, **String** field, **String** value) {  
 **return** *jedisCluster*.hset(key, field, value);  
 }  
  
 /\*\*  
 \* 将哈希表 key 中的域 field 的值设置为 value ，当且仅当域 field 不存在  
 \* 若域 field 已经存在，该操作无效  
 \* 如果 key 不存在，一个新哈希表被创建并执行 HSETNX 命令  
 \*  
 \* @param key  
 \* @param field  
 \* @param value  
 \* @return 设置成功，返回 1 ,如果给定域已经存在且没有操作被执行，返回 0  
 \*/  
 **public Long** hsetnx(**String** key, **String** field, **String** value) {  
 **return** *jedisCluster*.hsetnx(key, field, value);  
 }  
  
 /\*\*  
 \* 返回哈希表 key 中所有域的值  
 \*  
 \* @param key  
 \* @return 一个包含哈希表中所有值的表, 当 key 不存在时，返回一个空表  
 \*/  
 **public** List<**String**> hvals(**String** key) {  
 **return** *jedisCluster*.hvals(key);  
 }  
 //List(列表)  
  
 /\*\*  
 \* LPOP 命令的阻塞版本，当给定列表内没有任何元素可供弹出的时候，连接将被 BLPOP 命令阻塞，直到等待超时或发现可弹出元素为止  
 \*  
 \* @param timeout 超时时间，设置为0 表示无限制等待  
 \* @param key  
 \* @return  
 \*/  
 **public String** blpop(**int** timeout, **String** key) {  
 List<**String**> value = *jedisCluster*.blpop(timeout, key);  
 **return** value.get(**0**);  
 }  
  
 /\*\*  
 \* RPOP 命令的阻塞版本，当给定列表内没有任何元素可供弹出的时候，连接将被 BLPOP 命令阻塞，直到等待超时或发现可弹出元素为止  
 \*  
 \* @param timeout 超时时间，设置为0 表示无限制等待  
 \* @param key  
 \* @return  
 \*/  
 **public String** brpop(**int** timeout, **String** key) {  
 List<**String**> value = *jedisCluster*.brpop(timeout, key);  
 **return** value.get(**0**);  
 }  
  
 /\*\*  
 \* 返回列表 key 中，下标为 index 的元素。  
 \* 下标(index)参数 start 和 stop 都以 0 为底，也就是说，以 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  
 \* 如果 key 不是列表类型，返回一个错误  
 \*  
 \* @param key  
 \* @param index  
 \* @return  
 \*/  
 **public String** lindex(**String** key, **long** index) {  
 **return** *jedisCluster*.lindex(key, index);  
 }  
  
 /\*\*  
 \* 将值 value 插入到列表 key 当中，位于值 pivot 之前或之后  
 \* 当 pivot 不存在于列表 key 时，不执行任何操作  
 \* 当 key 不存在时， key 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  
 \*  
 \* @param key  
 \* @param where 值为BEFORE或者AFTER  
 \* @param pivot  
 \* @param value  
 \* @return 成功：返回list长度 无pivot：返回-1 key不存在或者空列表：返回0 where输入错误：返回-2  
 \*/  
 **public long** linsert(**String** key, **String** where, **String** pivot, **String** value) {  
 **long** result = -**2**;  
 **if** (where.toUpperCase() == "BEFORE") {  
 result = *jedisCluster*.linsert(key, LIST\_POSITION.BEFORE, pivot, value);  
 } **else if** (where.toUpperCase() == "AFTER") {  
 result = *jedisCluster*.linsert(key, LIST\_POSITION.AFTER, pivot, value);  
 }  
 **return** result;  
 }  
  
 /\*\*  
 \* 返回列表 key 的长度  
 \* 如果 key 不存在，则 key 被解释为一个空列表，返回 0  
 \* 如果 key 不是列表类型，返回一个错误  
 \*  
 \* @param key  
 \* @return  
 \*/  
 **public long** llen(**String** key) {  
 **return** *jedisCluster*.llen(key);  
 }  
  
 /\*\*  
 \* 移除并返回列表 key 的头元素  
 \*  
 \* @param key  
 \* @return  
 \*/  
 **public String** lpop(**String** key) {  
 **return** *jedisCluster*.lpop(key);  
 }  
  
 /\*\*  
 \* 将一个或多个值 value 插入到列表 key 的表头  
 \* 如果 key 不存在，一个空列表会被创建并执行 LPUSH 操作  
 \* 当 key 存在但不是列表类型时，返回一个错误  
 \*  
 \* @param key  
 \* @param values 要插入的value的组合，格式为：value,value,......  
 \* @return 列表长度  
 \*/  
 **public long** lpush(**String** key, **String** values) {  
 **return** *jedisCluster*.lpush(key, values);  
 }  
  
 /\*\*  
 \* 将值 value 插入到列表 key 的表头  
 \*  
 \* @param key  
 \* @param value  
 \* @return 列表长度  
 \*/  
 **public long** lpushx(**String** key, **String** value) {  
 **return** *jedisCluster*.lpushx(key, value);  
 }  
  
 /\*\*  
 \* 返回列表 key 中指定区间内的元素，区间以偏移量 start 和 stop 指定  
 \* 下标(index)参数 start 和 stop 都以 0 为底，也就是说，以 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。  
 \*  
 \* @param key  
 \* @param start  
 \* @param stop  
 \* @return  
 \*/  
 **public** List<**String**> lrange(**String** key, **long** start, **long** stop) {  
 **return** *jedisCluster*.lrange(key, start, stop);  
 }  
  
 /\*\*  
 \* 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  
 \*  
 \* @param key  
 \* @param count  
 \* @param value  
 \* @return 被移除元素的数量  
 \*/  
 **public long** lrem(**String** key, **long** count, **String** value) {  
 **return** *jedisCluster*.lrem(key, count, value);  
 }  
  
 /\*\*  
 \* 将列表 key 下标为 index 的元素的值设置为 value  
 \* 当 index 参数超出范围，或对一个空列表( key 不存在)进行 LSET 时，返回一个错误  
 \*  
 \* @param key  
 \* @param index  
 \* @param value  
 \* @return  
 \*/  
 **public String** lset(**String** key, **long** index, **String** value) {  
 **return** *jedisCluster*.lset(key, index, value);  
 }  
  
 /\*\*  
 \* 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  
 \*  
 \* @param key  
 \* @param start  
 \* @param end  
 \* @return 成功返回OK  
 \*/  
 **public String** ltrim(**String** key, **long** start, **long** end) {  
 **return** *jedisCluster*.ltrim(key, start, end);  
 }  
  
 /\*\*  
 \* 移除并返回列表 key 的尾元素  
 \* 当 key 不存在时，返回 nil  
 \*  
 \* @param key  
 \* @return  
 \*/  
 **public String** rpop(**String** key) {  
 **return** *jedisCluster*.rpop(key);  
 }  
  
 /\*\*  
 \* 将一个或多个值 value 插入到列表 key 的表尾  
 \* 如果 key 不存在，一个空列表会被创建并执行 RPUSH 操作  
 \* 当 key 存在但不是列表类型时，返回一个错误  
 \*  
 \* @param key  
 \* @param values 要插入的value的组合，格式为：value,value,......  
 \* @return 列表长度  
 \*/  
 **public long** rpush(**String** key, **String** values) {  
 **return** *jedisCluster*.rpush(key, values);  
 }  
  
 /\*\*  
 \* 将值 value 插入到列表 key 的表尾  
 \*  
 \* @param key  
 \* @param value  
 \* @return 列表长度  
 \*/  
 **public long** rpushx(**String** key, **String** value) {  
 **return** *jedisCluster*.rpushx(key, value);  
 }  
 //Set(集合)  
  
 /\*\*  
 \* 将一个或多个 member 元素加入到集合 key 当中，已经存在于集合的 member 元素将被忽略  
 \* 假如 key 不存在，则创建一个只包含 member 元素作成员的集合  
 \* 当 key 不是集合类型时，返回一个错误  
 \*  
 \* @param key  
 \* @param members 格式为：member,member,.......  
 \* @return 被添加到集合中的新元素的数量，不包括被忽略的元素  
 \*/  
 **public long** sadd(**String** key, **String** members) {  
 **return** *jedisCluster*.sadd(key, members);  
 }  
  
 /\*\*  
 \* 返回集合 key 的基数(集合中元素的数量)  
 \*  
 \* @param key  
 \* @return 集合的基数, 当 key 不存在时，返回 0  
 \*/  
 **public long** scard(**String** key) {  
 **return** *jedisCluster*.scard(key);  
 }  
  
 /\*\*  
 \* 判断 member 元素是否集合 key 的成员  
 \*  
 \* @param key  
 \* @param member  
 \* @return 如果 member 元素是集合的成员，返回 1,如果 member 元素不是集合的成员，或 key 不存在，返回 0  
 \*/  
 **public Boolean** sismember(**String** key, **String** member) {  
 **return** *jedisCluster*.sismember(key, member);  
 }  
  
 /\*\*  
 \* 返回集合 key 中的所有成员,不存在的 key 被视为空集合  
 \*  
 \* @param key  
 \* @return 集合中的所有成员  
 \*/  
 **public** Set<**String**> smembers(**String** key) {  
 **return** *jedisCluster*.smembers(key);  
 }  
  
 /\*\*  
 \* 移除并返回集合中的一个随机元素  
 \*  
 \* @param key  
 \* @return 被移除的随机元素, 当 key 不存在或 key 是空集时，返回 nil  
 \*/  
 **public String** spop(**String** key) {  
 **return** *jedisCluster*.spop(key);  
 }  
  
 /\*\*  
 \* 移除并返回集合中的一个随机元素  
 \*  
 \* @param key  
 \* @return 被移除的随机元素, 当 key 不存在或 key 是空集时，返回 nil  
 \*/  
 **public** Set<**String**> spop(**String** key, **long** count) {  
 **return** *jedisCluster*.spop(key, count);  
 }  
  
 /\*\*  
 \* 只提供了 key 参数，那么返回集合中的一个随机元素, 不做删除  
 \*  
 \* @param key  
 \* @return 返回一个元素；如果集合为空，返回 nil  
 \*/  
 **public String** srandMember(**String** key) {  
 **return** *jedisCluster*.srandmember(key);  
 }  
  
 /\*\*  
 \* 如果 count 为正数，且小于集合基数，那么命令返回一个包含 count 个元素的数组，数组中的元素各不相同。如果 count 大于等于集合基数，那么返回整个集合  
 \* 如果 count 为负数，那么命令返回一个数组，数组中的元素可能会重复出现多次，而数组的长度为 count 的绝对值  
 \*  
 \* @param key  
 \* @param count  
 \* @return 返回一个数组；如果集合为空，返回空数组  
 \*/  
 **public** List<**String**> srandMember(**String** key, **int** count) {  
 **return** *jedisCluster*.srandmember(key, count);  
 }  
  
 /\*\*  
 \* 返回一个数组；如果集合为空，返回空数组  
 \*  
 \* @param key  
 \* @param members 格式为：member,member,......  
 \* @return 被成功移除的元素的数量，不包括被忽略的元素  
 \*/  
 **public long** srem(**String** key, **String** members) {  
 **return** *jedisCluster*.srem(key, members);  
 }  
 //SortedSet(有序集合)  
  
 /\*\*  
 \* 将一个member 元素及其 score 值加入到有序集 key 当中  
 \* 如果某个 member 已经是有序集的成员，那么更新这个 member 的 score 值，并通过重新插入这个 member 元素，来保证该 member 在正确的位置上  
 \*  
 \* @param key  
 \* @param score  
 \* @param member  
 \* @return 被成功添加的新成员的数量，不包括那些被更新的、已经存在的成员  
 \*/  
 **public long** zadd(**String** key, **double** score, **String** member) {  
 **return** *jedisCluster*.zadd(key, score, member);  
 }  
  
 /\*\*  
 \* 将一个或多个 member 元素及其 score 值加入到有序集 key 当中  
 \* 如果某个 member 已经是有序集的成员，那么更新这个 member 的 score 值，并通过重新插入这个 member 元素，来保证该 member 在正确的位置上  
 \*  
 \* @param key  
 \* @param scoreMembers 格式为：member,score,member,score,......  
 \* @return 被成功添加的新成员的数量，不包括那些被更新的、已经存在的成员  
 \*/  
 **public long** zadd(**String** key, **String** scoreMembers) {  
 **String**[] scoreMember = scoreMembers.split(",");  
 Map<**String**, **Double**> map = **new** HashMap<**String**, **Double**>();  
 **for** (**int** i = **0**; i < scoreMember.*length*; i++) {  
 map.put(scoreMember[i], **Double**.valueOf(scoreMember[i + **1**]));  
 }  
 **return** *jedisCluster*.zadd(key, map);  
 }  
  
 /\*\*  
 \* 返回有序集 key 的基数  
 \*  
 \* @param key  
 \* @return 当 key 存在且是有序集类型时，返回有序集的基数,当 key 不存在时，返回 0  
 \*/  
 **public long** zcard(**String** key) {  
 **return** *jedisCluster*.zcard(key);  
 }  
  
 /\*\*  
 \* 返回有序集 key 中， score 值在 min 和 max 之间(默认包括 score 值等于 min 或 max )的成员的数量  
 \*  
 \* @param key  
 \* @param min  
 \* @param max  
 \* @return score 值在 min 和 max 之间的成员的数量  
 \*/  
 **public long** zcount(**String** key, **double** min, **double** max) {  
 **return** *jedisCluster*.zcount(key, min, max);  
 }  
  
 /\*\*  
 \* 为有序集 key 的成员 member 的 score 值加上增量 increment  
 \* 可以通过传递一个负数值 increment ，让 score 减去相应的值，比如 ZINCRBY key -5 member ，就是让 member 的 score 值减去 5  
 \* 当 key 不存在，或 member 不是 key 的成员时， ZINCRBY key increment member 等同于 ZADD key increment member  
 \* 当 key 不是有序集类型时，返回一个错误  
 \*  
 \* @param key  
 \* @param score  
 \* @param member  
 \* @return member 成员的新 score 值  
 \*/  
 **public double** zincrby(**String** key, **double** increment, **String** member) {  
 **return** *jedisCluster*.zincrby(key, increment, member);  
 }  
  
 /\*\*  
 \* 返回有序集 key 中，指定区间内的成员,其中成员的位置按 score 值递增(从小到大)来排序  
 \* 具有相同 score 值的成员按字典序(lexicographical order )来排列  
 \* 下标参数 start 和 stop 都以 0 为底，也就是说，以 0 表示有序集第一个成员，以 1 表示有序集第二个成员，以此类推。  
 \* 你也可以使用负数下标，以 -1 表示最后一个成员， -2 表示倒数第二个成员，以此类推  
 \*  
 \* @param key  
 \* @param start  
 \* @param end  
 \* @return 指定区间内，带有 score 值(可选)的有序集成员的列表  
 \*/  
 **public** Set<**String**> zrange(**String** key, **long** start, **long** end) {  
 **return** *jedisCluster*.zrange(key, start, end);  
 }  
  
 /\*\*  
 \* 返回有序集 key 中，所有 score 值介于 min 和 max 之间(包括等于 min 或 max )的成员。有序集成员按 score 值递增(从小到大)次序排列  
 \*  
 \* @param key  
 \* @param min  
 \* @param max  
 \* @return 指定区间内，带有 score 值(可选)的有序集成员的列表  
 \*/  
 **public** Set<**String**> zrangeByScore(**String** key, **String** min, **String** max) {  
 **return** *jedisCluster*.zrangeByScore(key, min, max);  
 }  
  
 /\*\*  
 \* 返回有序集 key 中，所有 score 值介于 min 和 max 之间(包括等于 min 或 max )的成员。有序集成员按 score 值递增(从小到大)次序排列  
 \*  
 \* @param key  
 \* @param min  
 \* @param max  
 \* @return 指定区间内，带有 score 值(可选)的有序集成员的列表  
 \*/  
 **public** Set<**String**> zrangeByScore(**String** key, **double** min, **double** max) {  
 **return** *jedisCluster*.zrangeByScore(key, min, max);  
 }  
  
 /\*\*  
 \* 返回有序集 key 中，所有 score 值介于 min 和 max 之间(包括等于 min 或 max )的成员。有序集成员按 score 值递增(从小到大)次序排列  
 \* 可选的 LIMIT 参数指定返回结果的数量及区间(就像SQL中的 SELECT LIMIT offset, count )，注意当 offset 很大时，定位 offset 的操作可能需要遍历整个有序集，此过程最坏复杂度为 O(N) 时间  
 \*  
 \* @param key  
 \* @param min  
 \* @param max  
 \* @return 指定区间内，带有 score 值(可选)的有序集成员的列表  
 \*/  
 **public** Set<**String**> zrangeByScore(**String** key, **String** min, **String** max, **int** offset, **int** count) {  
 **return** *jedisCluster*.zrangeByScore(key, min, max, offset, count);  
 }  
  
 /\*\*  
 \* 返回有序集 key 中，所有 score 值介于 min 和 max 之间(包括等于 min 或 max )的成员。有序集成员按 score 值递增(从小到大)次序排列  
 \* 可选的 LIMIT 参数指定返回结果的数量及区间(就像SQL中的 SELECT LIMIT offset, count )，注意当 offset 很大时，定位 offset 的操作可能需要遍历整个有序集，此过程最坏复杂度为 O(N) 时间  
 \*  
 \* @param key  
 \* @param min  
 \* @param max  
 \* @return 指定区间内，带有 score 值(可选)的有序集成员的列表  
 \*/  
 **public** Set<**String**> zrangeByScore(**String** key, **double** min, **double** max, **int** offset, **int** count) {  
 **return** *jedisCluster*.zrangeByScore(key, min, max, offset, count);  
 }  
  
 /\*\*  
 \* 返回有序集 key 中成员 member 的排名。其中有序集成员按 score 值递增(从小到大)顺序排列  
 \* 排名以 0 为底，也就是说， score 值最小的成员排名为 0  
 \*  
 \* @param key  
 \* @param member  
 \* @return  
 \*/  
 **public long** zrank(**String** key, **String** member) {  
 **return** *jedisCluster*.zrank(key, member);  
 }  
  
 /\*\*  
 \* 移除有序集 key 中的一个或多个成员，不存在的成员将被忽略  
 \* 当 key 存在但不是有序集类型时，返回一个错误  
 \*  
 \* @param key  
 \* @param members 格式为：member,member,......  
 \* @return 被成功移除的成员的数量，不包括被忽略的成员  
 \*/  
 **public long** zrem(**String** key, **String** members) {  
 **return** *jedisCluster*.zrem(key, members);  
 }  
  
 /\*\*  
 \* 移除有序集 key 中，指定排名(rank)区间内的所有成员  
 \* 区间分别以下标参数 start 和 stop 指出，包含 start 和 stop 在内  
 \*  
 \* @param key  
 \* @param start  
 \* @param end  
 \* @return 被移除成员的数量  
 \*/  
 **public long** zremRangeByRank(**String** key, **long** start, **long** end) {  
 **return** *jedisCluster*.zremrangeByRank(key, start, end);  
 }  
  
 /\*\*  
 \* 移除有序集 key 中，所有 score 值介于 min 和 max 之间(包括等于 min 或 max )的成员。  
 \*  
 \* @param key  
 \* @param start  
 \* @param end  
 \* @return 被移除成员的数量  
 \*/  
 **public long** zremRangeByScore(**String** key, **String** start, **String** end) {  
 **return** *jedisCluster*.zremrangeByScore(key, start, end);  
 }  
  
 /\*\*  
 \* 移除有序集 key 中，所有 score 值介于 min 和 max 之间(包括等于 min 或 max )的成员。  
 \*  
 \* @param key  
 \* @param start  
 \* @param end  
 \* @return 被移除成员的数量  
 \*/  
 **public long** zremRangeByScore(**String** key, **double** start, **double** end) {  
 **return** *jedisCluster*.zremrangeByScore(key, start, end);  
 }  
  
 /\*\*  
 \* 返回有序集 key 中，指定区间内的成员  
 \* 其中成员的位置按 score 值递减(从大到小)来排列  
 \*  
 \* @param key  
 \* @param start  
 \* @param end  
 \* @return 指定区间内，带有 score 值(可选)的有序集成员的列表  
 \*/  
 **public** Set<**String**> zrevRange(**String** key, **long** start, **long** end) {  
 **return** *jedisCluster*.zrevrange(key, start, end);  
 }  
  
 /\*\*  
 \* 返回有序集 key 中，所有 score 值介于 max 和 min 之间(包括等于 max 或 min )的成员。有序集成员按 score 值递减(从大到小)次序排列  
 \*  
 \* @param key  
 \* @param min  
 \* @param max  
 \* @return 指定区间内，带有 score 值(可选)的有序集成员的列表  
 \*/  
 **public** Set<**String**> zrevrangeByScore(**String** key, **String** max, **String** min) {  
 **return** *jedisCluster*.zrevrangeByScore(key, max, min);  
 }  
  
 /\*\*  
 \* 返回有序集 key 中，所有 score 值介于 max 和 min 之间(包括等于 max 或 min )的成员。有序集成员按 score 值递减(从大到小)次序排列  
 \*  
 \* @param key  
 \* @param min  
 \* @param max  
 \* @return 指定区间内，带有 score 值(可选)的有序集成员的列表  
 \*/  
 **public** Set<**String**> zrevrangeByScore(**String** key, **double** max, **double** min) {  
 **return** *jedisCluster*.zrevrangeByScore(key, max, min);  
 }  
  
 /\*\*  
 \* 返回有序集 key 中，所有 score 值介于 max 和 min 之间(包括等于 max 或 min )的成员。有序集成员按 score 值递减(从大到小)次序排列  
 \* 可选的 LIMIT 参数指定返回结果的数量及区间(就像SQL中的 SELECT LIMIT offset, count )，注意当 offset 很大时，定位 offset 的操作可能需要遍历整个有序集，此过程最坏复杂度为 O(N) 时间  
 \*  
 \* @param key  
 \* @param min  
 \* @param max  
 \* @return 指定区间内，带有 score 值(可选)的有序集成员的列表  
 \*/  
 **public** Set<**String**> zrevrangeByScore(**String** key, **String** max, **String** min, **int** offset, **int** count) {  
 **return** *jedisCluster*.zrevrangeByScore(key, max, min, offset, count);  
 }  
  
 /\*\*  
 \* 返回有序集 key 中，所有 score 值介于 max 和 min 之间(包括等于 max 或 min )的成员。有序集成员按 score 值递减(从大到小)次序排列  
 \* 可选的 LIMIT 参数指定返回结果的数量及区间(就像SQL中的 SELECT LIMIT offset, count )，注意当 offset 很大时，定位 offset 的操作可能需要遍历整个有序集，此过程最坏复杂度为 O(N) 时间  
 \*  
 \* @param key  
 \* @param min  
 \* @param max  
 \* @return 指定区间内，带有 score 值(可选)的有序集成员的列表  
 \*/  
 **public** Set<**String**> zrevrangeByScore(**String** key, **double** max, **double** min, **int** offset, **int** count) {  
 **return** *jedisCluster*.zrevrangeByScore(key, max, min, offset, count);  
 }  
  
 /\*\*  
 \* 返回有序集 key 中成员 member 的排名。其中有序集成员按 score 值递减(从大到小)排序  
 \* 排名以 0 为底，也就是说， score 值最大的成员排名为 0  
 \*  
 \* @param key  
 \* @param member  
 \* @return 如果 member 是有序集 key 的成员，返回 member 的排名,如果 member 不是有序集 key 的成员，返回 nil  
 \*/  
 **public long** zrevRank(**String** key, **String** member) {  
 **return** *jedisCluster*.zrevrank(key, member);  
 }  
  
 /\*\*  
 \* 返回有序集 key 中，成员 member 的 score 值  
 \* 如果 member 元素不是有序集 key 的成员，或 key 不存在，返回 nil  
 \*  
 \* @param key  
 \* @param member  
 \* @return member 成员的 score 值  
 \*/  
 **public Double** zscore(**String** key, **String** member) {  
 **return** *jedisCluster*.zscore(key, member);  
 }  
 //Connection(连接)  
  
 /\*\*  
 \* 通过设置配置文件中 requirepass 项的值(使用命令 CONFIG SET requirepass password )，可以使用密码来保护 Redis 服务器  
 \* 如果开启了密码保护的话，在每次连接 Redis 服务器之后，就要使用 AUTH 命令解锁，解锁之后才能使用其他 Redis 命令  
 \* 如果 AUTH 命令给定的密码 password 和配置文件中的密码相符的话，服务器会返回 OK 并开始接受命令输入  
 \* 另一方面，假如密码不匹配的话，服务器将返回一个错误，并要求客户端需重新输入密码。  
 \*  
 \* @param password  
 \* @return 密码匹配时返回 OK ，否则返回一个错误  
 \*/  
 **public String** auth(**String** password) {  
 **return** *jedisCluster*.auth(password);  
 }  
  
 /\*\*  
 \* 打印一个特定的信息 message ，测试时使用  
 \*  
 \* @param message  
 \* @return message 自身  
 \*/  
 **public String** echo(**String** message) {  
 **return** *jedisCluster*.echo(message);  
 }  
  
 /\*\*  
 \* 使用客户端向 Redis 服务器发送一个 PING ，如果服务器运作正常的话，会返回一个 PONG  
 \* 通常用于测试与服务器的连接是否仍然生效，或者用于测量延迟值  
 \*  
 \* @return 如果连接正常就返回一个 PONG ，否则返回一个连接错误  
 \*/  
 **public String** ping() {  
 **return** *jedisCluster*.ping();  
 }  
  
 /\*\*  
 \* 请求服务器关闭与当前客户端的连接  
 \* 一旦所有等待中的回复(如果有的话)顺利写入到客户端，连接就会被关闭  
 \*  
 \* @return 总是返回 OK  
 \*/  
 **public String** quit() {  
 **return** *jedisCluster*.quit();  
 }  
  
 /\*\*  
 \* 切换到指定的数据库，数据库索引号 index 用数字值指定，以 0 作为起始索引值,默认使用 0 号数据库  
 \*  
 \* @param index  
 \* @return ok  
 \*/  
 **public String** select(**int** index) {  
 **return** *jedisCluster*.select(index);  
 }  
 //Server(服务器)  
  
 /\*\*  
 \* 执行一个 AOF文件 重写操作。重写会创建一个当前 AOF 文件的体积优化版本  
 \*  
 \* @return 反馈信息  
 \*/  
 **public String** bgreWriteAof() {  
 **return** *jedisCluster*.bgrewriteaof();  
 }  
  
 /\*\*  
 \* 在后台异步(Asynchronously)保存当前数据库的数据到磁盘  
 \* BGSAVE 命令执行之后立即返回 OK ，然后 Redis fork 出一个新子进程，原来的 Redis 进程(父进程)继续处理客户端请求，而子进程则负责将数据保存到磁盘，然后退出  
 \*  
 \* @return 反馈信息  
 \*/  
 **public String** bgSave() {  
 **return** *jedisCluster*.bgsave();  
 }  
  
 /\*\*  
 \* 返回当前数据库的 key 的数量  
 \*  
 \* @return 当前数据库的 key 的数量  
 \*/  
 **public long** dbSize() {  
 **return** *jedisCluster*.dbSize();  
 }  
  
 /\*\*  
 \* 清空整个 Redis 服务器的数据(删除所有数据库的所有 key )  
 \*  
 \* @return 总是返回 OK  
 \*/  
 **public String** flushAll() {  
 **return** *jedisCluster*.flushAll();  
 }  
  
 /\*\*  
 \* 清空当前数据库中的所有 key  
 \*  
 \* @return 总是返回 OK  
 \*/  
 **public String** flushDB() {  
 **return** *jedisCluster*.flushDB();  
 }  
  
 /\*\*  
 \* 返回关于 Redis 服务器的各种信息和统计数值  
 \*  
 \* @return  
 \*/  
 **public String** info() {  
 **return** *jedisCluster*.info();  
 }  
  
 /\*\*  
 \* 以一种易于解释（parse）且易于阅读的格式，返回关于 Redis 服务器的各种信息和统计数值  
 \*  
 \* @param section  
 \* @return  
 \*/  
 **public String** info(**String** section) {  
 **return** *jedisCluster*.info(section);  
 }  
  
 /\*\*  
 \* 返回最近一次 Redis 成功将数据保存到磁盘上的时间，以 UNIX 时间戳格式表示  
 \*  
 \* @return  
 \*/  
 **public long** lastsave() {  
 **return** *jedisCluster*.lastsave();  
 }  
  
 /\*\*  
 \* SAVE 命令执行一个同步保存操作，将当前 Redis 实例的所有数据快照(snapshot)以 RDB 文件的形式保存到硬盘  
 \*  
 \* @return 保存成功时返回 OK  
 \*/  
 **public String** save() {  
 **return** *jedisCluster*.save();  
 }  
  
 /\*\*  
 \* SHUTDOWN 命令执行以下操作：停止所有客户端  
 \* 如果有至少一个保存点在等待，执行 SAVE 命令  
 \* 如果 AOF 选项被打开，更新 AOF 文件  
 \* 关闭 redis 服务器(server)  
 \* 如果持久化被打开的话， SHUTDOWN 命令会保证服务器正常关闭而不丢失任何数据  
 \*  
 \* @return 执行失败时返回错误, 执行成功时不返回任何信息，服务器和客户端的连接断开，客户端自动退出  
 \*/  
 **public String** shutdown() {  
 **return** *jedisCluster*.shutdown();  
 }  
  
 /\*\*  
 \* SLAVEOF 命令用于在 Redis 运行时动态地修改复制(replication)功能的行为  
 \* 通过执行 SLAVEOF host port 命令，可以将当前服务器转变为指定服务器的从属服务器(slave server)  
 \*  
 \* @param host  
 \* @param port  
 \* @return 总是返回 OK  
 \*/  
 **public String** slaveof(**String** host, **int** port) {  
 **return** *jedisCluster*.slaveof(host, port);  
 }  
}

## Jedis事务

**public** **class** TestTransaction{  
   **public** **boolean** transMethod(){  
      Jedis jedis= **new** Jedis("127.0.0.1",**6379**);  
      **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");  
        **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);  
        **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){  
      TestTransaction test= **new** TestTransaction();  
      **boolean** retValue=test.transMethod();  
      System.out.println("main retValue-------: " +retValue);  
   }  
 }

## Jedis主从复制

**public** **static** **void** main(String[] args) **throws** InterruptedException   
   {  
      Jedis jedis\_M = **new** Jedis("127.0.0.1",**6379**);  
      Jedis jedis\_S = **new** Jedis("127.0.0.1",**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\_Pool

**package** com.atguigu.redis.test;  
  
  
**import** redis.clients.jedis.Jedis;  
**import** redis.clients.jedis.JedisPool;  
**import** redis.clients.jedis.JedisPoolConfig;  
  
  
**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.setMaxActive(**1000**);  
            poolConfig.setMaxIdle(**32**);  
            poolConfig.setMaxWait(**100**\***1000**);  
            poolConfig.setTestOnBorrow(**true**);   
            jedisPool = **new** JedisPool(poolConfig,"127.0.0.1");  
          }  
       }  
     }  
      **return** jedisPool;  
  }  
   **public** **static** **void** release(JedisPool jedisPool,Jedis jedis) {  
      **if**(**null** != jedis)  
     {  
       jedisPool.returnResourceObject(jedis);  
     }  
 }  
 }

## RedisPool配置总结

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