事务有四个特性：

1. 原子性(Atomicity)

事务里的操作要么全部做完，要么都不做，事务成功的条件是事务里的所有操作都成功，只要有一个操作失败，整个事务就失败，需要回滚。

1. 一致性(Consistency)

数据库要处于一致的状态，事务的运行不会改变数据库原本的一致性约束

1. 独立性(Isolation)

独立性是指并发的事务之间不会相互影响，如果一个事务要访问的数据正在被另外一个事务修改，只要另外一个事务未提交，它所访问的数据就不受未提交事务的影响

1. 持久性(Durability)

持久性是指一旦事务提交后，它所做的修改将会永久的保存在数据库上，即使出现宕机，也不会丢失。

分布式：在不同的多台服务器上部署不同的服务模块（工程），他们之间通过rpc/rmi之间通信和调用，对外提供服务和组内协作。

集群：不同的多台服务器上面部署相同的服务模块，通过分布式调度软件进行统一的调度，对外提供服务和访问。

redis命令大全：http://redisdoc.com/

redis值可以使用的数据结构：

字符串、列表、散列、集合、有序集合、HyperLogLog

String类型：最基本的类型，一个key对应一个value；最多512M

hash：redis hash 是一个键值对集合，是一个string类型的field和value的映射表，hash特别适用于存储对象。类似于java中的Map<String,Object>.

list(列表)：列表是简单的字符串列表，按照插入顺序排序，可以添加一个元素导列表的头部（左边）或者尾部（右边）。他的底层是一个链表。

set（无序集合）：通过hashtable实现。

Zset（有序集合）：zset和set一样也是String类型的集合，且不允许重复的成员。redis通过分数来为集合中的成员进行排序。不同的每个成员都关联一个double类型的分数。zset的成员是唯一的，但分数却可以重复

redis的附加功能：

1. 持久化功能：将存储在内存里面的数据保存到硬盘里，保障数据安全，方便进行数据备份和恢复。
2. 发布与订阅功能：将消息同时分发给多个客户端，用于构建广播系统。
3. 过期键值功能：为键设置一个过期时间，让它在指定的时间之后自动删除。
4. 事务功能：原子地执行多个操作，并提供乐观锁功能，保证处理数据时的安全性。
5. 脚本功能：在服务器端原子地执行多个操作，完成复杂的功能，并减少客户端与服务器之间的通讯往返次数。
6. 复制：为指定的redis服务器创建一个或多个复制品，用于提升数据安全性，分担读请求的负载。
7. sentinel：监控redis服务器的状态，并在服务器发生故障时，进行自动故障转移。
8. 集群：创建分布式数据库，每个服务器分别执行一部分写操作和读操作。

linux系统下，安装方法

解压redis文件之后，进入redis解压文件，输入make，自行安装redis文件

开启redis的方法：

进入redis目录，然后输入src/redis-server启动redis服务器

进入redis目录，然后输入src/redis-cli进入redis客户端

关闭redis

输入：shutdown

输入：exit

windows系统下，用doc系统，安装方法

解压redis文件之后，进入redis解压文件，输入src/redis-server启动redis服务器

进入redis目录，然后输入src/redis-cli进入redis客户端

nx和xx选项：

nx：命令仅在键key不存在的情况下，才进行设置操作；如果key键已经存在，那么set…nx命令不做动作（不会覆盖旧值）。

xx：命令仅在key键已存在的情况下，才进行设置操作；如果key键不存在，那么set…xx命令不做动作（一定会覆盖旧值）。

在给定nx选项xx选项的情况下，set命令在设置成功时返回ok，设置失败返回nil。

redis缓存：

|  |  |  |
| --- | --- | --- |
| api | 效果 | 实现 |
| Cache(client) | 设置缓存程序使用的客户端 |  |
| Cache.put(name,content) | 把指定的内容放到缓存里面，并使用name来命名它，以便之后取出。 | 调用set命令 |
| Cache.get(name) | 从缓存中取出以name命名的内容。 | 调用get命令 |

例如：

@app.route(“/”)

def index():

cached\_content = cache.get(‘index’) #尝试从缓存中获取被缓存的页面

if cached\_content: #缓存存在，直接返回页面

return cached\_content

else:

content = fetch\_and\_create\_index() #页面没有缓存，访问数据库并重新生成页面

cache.put(‘index’,content) #缓存页面，方便下次取出

return content #返回页面

setnx key value

仅在key键不存在的情况下，将键key的值设置为value，效果和set key value nx 一样。nx的意思是“not exists”

键不存在并且设置成功时，命令放回1；因为键已经存在而到时设置失败时，返回0

同时设置或获取多个字符串键的值

|  |  |
| --- | --- |
| 命令 | 效果 |
| mset key value [key value ...] | 一次为一个或多个字符串键设置值，效果和同时执行多个set命令一样，命令返回ok |
| mget key [key ...] | 一次返回一个或多个字符串键的值，效果和同时执行多个get命令一样。 |

例如：

mset huangz::email “[huangz1990@gmail.com](mailto:huangz1990@gmail.com)” huangz::homepage “<http://huangz.me/> ” huangz::company “FakeCompany” huangz::position “programmer” huangz::location “广东” huangz::sign “time waits for no one”

mget huangz::email huangz::homepage huangz::company huangz::position...

注：“：：”为分隔符。也可以用其他分隔符。

当有部分key存在的时候，msetnx命令同时失效。

一次设置多个不存在的键值：

msetnx key value [key value ...]

设置新值并返回旧值：getset key new-value

返回值的长度：strlen key

keys \* 查看本库有多少个键

exists key：判断某个key是否存在

expire key 秒：设置过期时间

setex key x（秒） value：设置key的存活时间和值

ttl key：查看还有多少秒过期,-1表示永不过期，-2表示已过期（time to live）

type key：查看key的类型

incrby key xxx：增加

decrby key xxx：减少

setrange key x（下标） xxxxx：在key中的x下标后面加上xxxxx内容

getrange key x y：获取key中从x到y的值（0 -1表示全部）

lpush listkey x x x：从后往前排序（left：从左往右进栈）

rpush listkey x x x：从前往后排序（right：从右往左进栈）

lrange listkey x y：取出listkey中的值（0 -1表示全部）

lpop listkey：从左边出栈（出栈之后，栈中消失）

rpop listkey：从右边出栈（出栈之后，栈中消失）

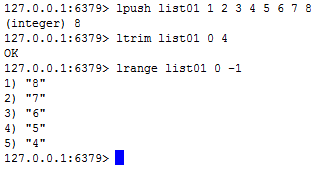
lindex x：按照索引下标获取元素

lindex list01 2：获取下标为2的值

lrem key n value：删除n个value

例：lrem list01 2 3：删除list01列中两个3；

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

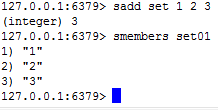


总结：

1. 它是一个字符串链表，left，right都可以插入添加，
2. 如果键不存在，创建新的链表
3. 如果键已存在，新增内容
4. 如果值全部移除，对应的键也就消失了
5. 链表的操作无论是头和尾的效率都极高，但是键入对中间的元素进行操作，效率很低

set散列

sadd setkey x y z



查看是否在set中：sismember setkey x



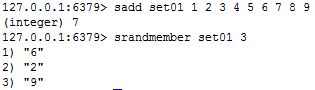
获取集合内的元素个数：scard setkey



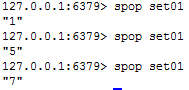
删除集合内元素：srem setkey x



随机出几个数字：srandmember setkey x



随机出栈：spop setkey（出一个）

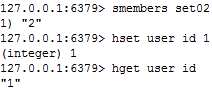


将set01里的值赋值给set02：smove set01 set02 2

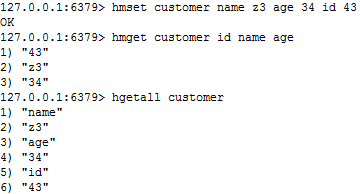


hash集合：kv模式不变，但是value是一个键值对

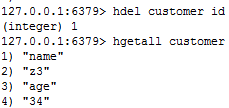
hset key value（key value）



存入多个hashset：hmset key value(key value)...



删除某个值：hdel key key



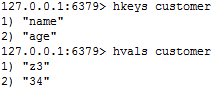
判断是否存在：hexists key key



显示长度：hlen key



获得hashset中的key或者value：hkeys/hvals key



若不存在，则插入，若存在则不插入：hsetnx key value(key value)



redis配置文件

include：包含些什么配置文件，可以是添加的。相当于web.xml或者mybatis中的mybatis-conf.xml

redis的安全：

config get requirepass：获取当前密码。

config set requirepass ‘123456’:设置redis密码为123456

auth 123456：在输入密码之后，才会连接成功redis。输入ping之后才会回pong，否则返回错误

常见redis配置

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

redis持久化

rdb:Redis DataBase

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中

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方式更加有效。RDB的缺点就是最后一次持久化后的数据可能丢失

rdb保存的是dump.rdb文件

禁用的方法：

save “”

开启的方法：

save <second> <changes>

save 900 1 //900秒内修改一次

save 300 10 //300秒内修改10次

save 60 1000 //60秒内修改1000次

一般使用：

一分钟修改1万次

5分钟修改10次

15分钟修改1次

优势:适合大规模的数据恢复, 对数据完整性和一致性要求不高

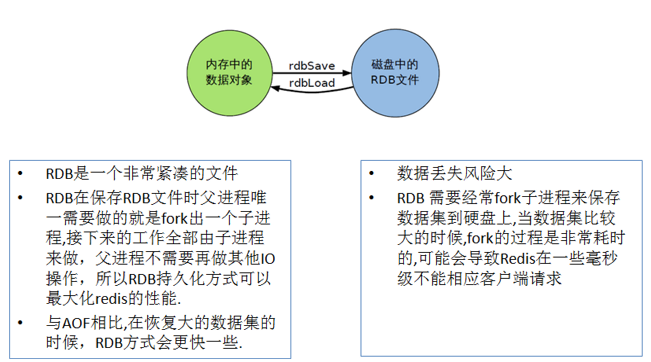
劣势：

在一定间隔时间做一次备份，所以如果redis意外down掉的话，就

会丢失最后一次快照后的所有修改。

fork的时候，内存中的数据被克隆了一份，大致2倍的膨胀性需要考虑

停止的方法：动态所有停止RDB保存规则的方法：redis-cli config set save ""

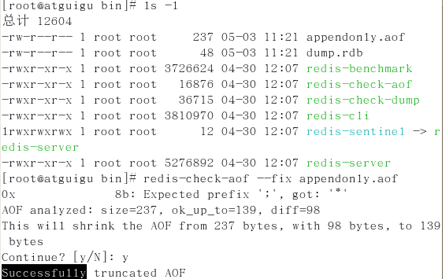


aof:Append Only File

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

Aof保存的是appendonly.aof文件

当aof文件出错时，需要使用修复命令



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

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

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

优点：

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

缺点：

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

持久化总结：

1. 当aof和rdb同时存在时，恢复数据的时候，访问的是aof文件
2. RDB持久化方式能够在指定的时间间隔能对你的数据进行快照存储

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

1. 只做缓存：如果你只希望你的数据在服务器运行的时候存在,你也可以不使用任何持久化方式.
2. 同时开启两种持久化方式：
   1. 在这种情况下,当redis重启的时候会优先载入AOF文件来恢复原始的数据,因为在通常情况下AOF文件保存的数据集要比RDB文件保存的数据集要完整.
   2. RDB的数据不实时，同时使用两者时服务器重启也只会找AOF文件。那要不要只使用AOF呢？作者建议不要，因为RDB更适合用于备份数据库(AOF在不断变化不好备份)，快速重启，而且不会有AOF可能潜在的bug，留着作为一个万一的手段。

建议：

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

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

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

redis的事务：

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

redis事务常用命令

discard：取消事务，放弃执行事务块内的所有命令；

exec：执行所有事务块内的命令；

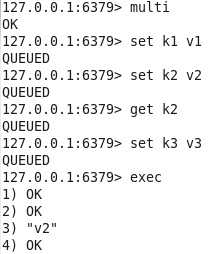
multi：标记一个事务块的开始；

unwatch：取消watch命令对所有key的监视；

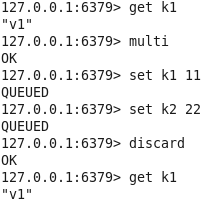
watch key[key ...]:监视一个（或多个）key，如果在事务执行之前这个（或这些）key被其他命令所改动，那么事务将被打断。

1. 正常执行

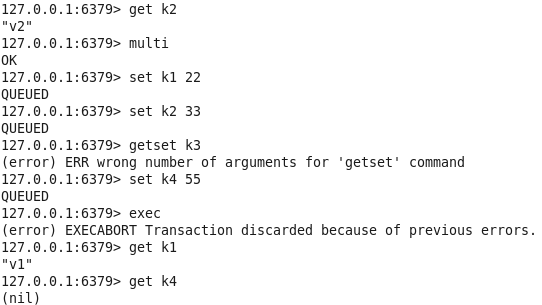
例如：



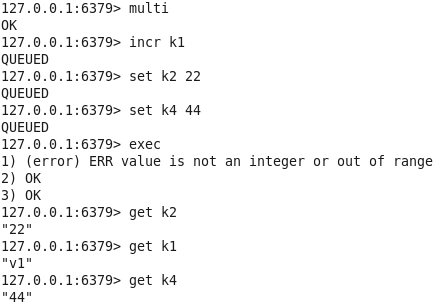
1. 放弃事务



1. 全体连坐：只要事务过程中出错，便全部不执行。命令在加入队列过程中就会报错



1. 冤头债主：命令加入队列过程中不报错，在执行过程中报错



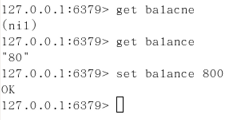
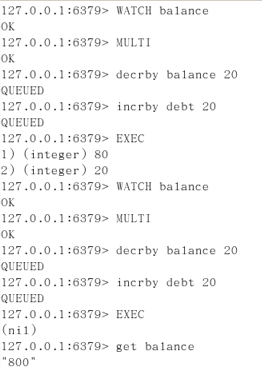
1. watch监控
   1. 乐观锁：每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制。乐观锁适用于多读的应用类型，这样可以提高吞吐量。

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

* 1. 悲观所：每次去拿数据的时候都会认为别人会修改，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面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。

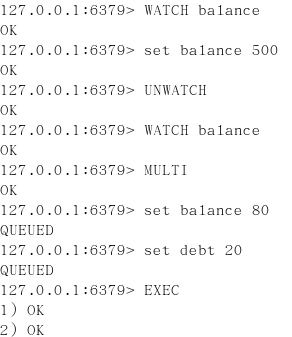
备份的时候。

* 1. CAS（check and set）



在执行watch balance之后，执行了set balance 800。会出现multi执行不成功的标识“nil”，查看balance的值，显示为800，表示在执行watch的过程中，有其他事务操作了此数据，版本号对不上，所以出现了事务执行不成功提示。

unwatch



小结：

1. 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已经被别的客户端push/pop过了，整个事务队列都不会执行
2. 通过watch命令，在事务执行之前监视多个key，若在watch之后有任何key的值发生变化，exec命令执行的事务都会被放弃，同时返回nullmulti-bulk应答以通知调用者事务执行失败。

redis事务的特性：

1. 单独的隔离操作：事务中的所有命令都会序列化，按顺序地执行。事务在执行过程中，不会被其他客户端发送过来的命令请求打断
2. 没有隔离级别的概念：队列中的命令没有提交之前都不会被执行，因为事务提交前任何指令都不会执行，也就不存在“事务内的查询要看事务里的更新，在事务外查询不能看到”这个问题。
3. 不保证原子性：同一个事务中如果有一条指令执行失败，其后的命令任然会被执行，没有回滚。

redis发布订阅

命令：

psubscribe pattem[pattem...]:订阅一个或多个符合给定模式的频道

pubsub subcommand [argument[argument...]]：查看订阅与发布系统

publish channel message：将消息发送到指定的频道

punsubscribe [pattem[pattem...]]：退订给定的一个或多个频道的信息

unsubscribe [channel [channel...]]：退订给定的频道

redis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/slave机制，master以写为主，slave以读为主

配置：配置从库不配置主库

方法：slaveof 主库ip 主库端口

每次和master断开之后，都需要重新连接，除非配置进redis.config文件

通过命令slaveof进行设置从机。

从机只能进行读命令，不能写。当主机死了之后，从机进入待命状态，不会变成主机。

slaveof no one:使当前数据库停止与其他数据库的同步，转成主库。

复制原理：slave启动成功连接到master后会发送一个sync命令，master接到命令启动后台的存盘进程，同时手机所有接收到的用于修改数据命令，在后台进程执行完毕后，master将传送整个数据文件到slave，以完成一次完全同步。

全量复制：而slave服务在接收到数据文件数据后，将其存盘并加载到内存中。

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

但是只要重新连接master，一次完全同步（全量复制）将被自动执行

哨兵模式：能够后台监控主机是否故障，如果故障了，根据投票数自动将从库转换为主库。

步骤：1.新建sentinel.conf文件

sentinel.conf:

sentinel monitor 被监控数据库名字（自己取名字） 本地ip（127.0.0.1） 6379 1

最后一个1表示：主机挂掉之后，slave投票看让谁阶梯称为主机，得票数为多少后称为主机

2.启动哨兵：

redis-sentinel /user/common/sentinel.conf(目录依照各自的实际情况配置，可能目录不同)

复制的缺点：由于所有的操作都是现在master上操作，然后同步更新到slave上，所以从master同步到slave机器有一定的延迟，当系统很繁忙的时候，延迟问题会更加严重，slave机器数量的增加也会使这个问题更加严重。

java中连接redis服务器

Jedis jedis = new Jedis("host(服务器地址)",端口);

开启事务：

Transaction transaction = jedis.multi();

……

transaction.exec();

transaction.discard();

加锁

**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);

  }

}

redis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