基础：

<http://www.runoob.com/redis/redis-tutorial.html>

<http://www.cnblogs.com/kaituorensheng/p/5244347.html>

<http://www.iigrowing.cn/redis-huan-cun-ji-shu-xue-xi.html>

<http://www.cnblogs.com/xiaoxi/category/961351.html>

<http://blog.csdn.net/column/details/13948.html>

http://www.cnblogs.com/zhaoguihua/category/584209.html

# 数据结构

## String类型

字符串类型是Redis中最基础的数据存储类型。

String类型在Redis中是二进制安全的，这便意味着该类型可以接受任何格式的数据，如JPEG图像数据或Json对象描述信息等。在Redis中字符串类型的Value最多可以容纳的数据长度是512M。

### 相关命令

1 SET key value

设置指定 key 的值

2 GET key

获取指定 key 的值。

3 GETRANGE key start end

返回 key 中字符串值的子字符

4 GETSET key value

将给定 key 的值设为 value ，并返回 key 的旧值(old value)。

5 GETBIT key offset

对 key 所储存的字符串值，获取指定偏移量上的位(bit)。

6 MGET key1 [key2..]

获取所有(一个或多个)给定 key 的值。

7 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)。

8 SETEX key seconds value

将值 value 关联到 key ，并将 key 的过期时间设为 seconds (以秒为单位)。

9 SETNX key value

只有在 key 不存在时设置 key 的值。

10 SETRANGE key offset value

用 value 参数覆写给定 key 所储存的字符串值，从偏移量 offset 开始。

11 STRLEN key

返回 key 所储存的字符串值的长度。

12 MSET key value [key value ...]

同时设置一个或多个 key-value 对。

13 MSETNX key value [key value ...]

同时设置一个或多个 key-value 对，当且仅当所有给定 key 都不存在。

14 PSETEX key milliseconds value

这个命令和 SETEX 命令相似，但它以毫秒为单位设置 key 的生存时间，而不是像 SETEX 命令那样，以秒为单位。

15 INCR key

将 key 中储存的数字值增一。

16 INCRBY key increment

将 key 所储存的值加上给定的增量值（increment） 。

17 INCRBYFLOAT key increment

将 key 所储存的值加上给定的浮点增量值（increment） 。

18 DECR key

将 key 中储存的数字值减一。

19 DECRBY key decrement

key 所储存的值减去给定的减量值（decrement） 。

20 APPEND key value

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。

## Hash类型

我们可以将Redis中的Hash类型看成具有String Key和String Value的map容器。所以该类型非常适合于存储值对象的信息。如Username、Password和Age等。如果Hash中包含很少的字段，那么该类型的数据也将仅占用很少的磁盘空间。每一个Hash可以存储4294967295个键值对。

### 相关命令

1 HDEL key field1 [field2]

删除一个或多个哈希表字段

2 HEXISTS key field

查看哈希表 key 中，指定的字段是否存在。

3 HGET key field

获取存储在哈希表中指定字段的值。

4 HGETALL key

获取在哈希表中指定 key 的所有字段和值

5 HINCRBY key field increment

为哈希表 key 中的指定字段的整数值加上增量 increment 。

6 HINCRBYFLOAT key field increment

为哈希表 key 中的指定字段的浮点数值加上增量 increment 。

7 HKEYS key

获取所有哈希表中的字段

8 HLEN key

获取哈希表中字段的数量

9 HMGET key field1 [field2]

获取所有给定字段的值

10 HMSET key field1 value1 [field2 value2 ]

同时将多个 field-value (域-值)对设置到哈希表 key 中。

11 HSET key field value

将哈希表 key 中的字段 field 的值设为 value 。

12 HSETNX key field value

只有在字段 field 不存在时，设置哈希表字段的值。

13 HVALS key

获取哈希表中所有值

14 HSCAN key cursor [MATCH pattern] [COUNT count]

迭代哈希表中的键值对。

## List类型

在Redis中，List类型是按照插入顺序排序的字符串链表。

和数据结构中的普通链表一样，我们可以在其头部(left)和尾部(right)添加新的元素。在插入时，如果该键并不存在，Redis将为该键创建一个新的链表。但是，如果链表中所有的元素均被移除，也就是list为空，不包含元素时，那么其对应的键也将会从数据库中删除。List中可以包含的最大元素数量是4294967295。

从元素插入和删除的效率视角来看，如果我们是在链表的两头插入或删除元素，这将会是非常高效的操作，即使链表中已经存储了百万条记录，该操作也可以在常量时间内完成。然而需要说明的是，如果元素插入或删除操作是作用于链表中间，那将会是非常低效的。

### 相关命令

1 BLPOP key1 [key2 ] timeout

移出并获取列表的第一个元素， 如果列表没有元素会阻塞列表直到等待超时或发现可弹出元素为止。

2 BRPOP key1 [key2 ] timeout

移出并获取列表的最后一个元素， 如果列表没有元素会阻塞列表直到等待超时或发现可弹出元素为止。

3 BRPOPLPUSH source destination timeout

从列表中弹出一个值，将弹出的元素插入到另外一个列表中并返回它； 如果列表没有元素会阻塞列表直到等待超时或发现可弹出元素为止。

4 LINDEX key index

通过索引获取列表中的元素

5 LINSERT key BEFORE|AFTER pivot value

在列表的元素前或者后插入元素

6 LLEN key

获取列表长度

7 LPOP key

移出并获取列表的第一个元素

8 LPUSH key value1 [value2]

将一个或多个值插入到列表头部

9 LPUSHX key value

将一个值插入到已存在的列表头部

10 LRANGE key start stop

获取列表指定范围内的元素

11 LREM key count value

移除列表元素

12 LSET key index value

通过索引设置列表元素的值

13 LTRIM key start stop

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。

14 RPOP key

移除并获取列表最后一个元素

15 RPOPLPUSH source destination

移除列表的最后一个元素，并将该元素添加到另一个列表并返回

16 RPUSH key value1 [value2]

在列表中添加一个或多个值

17 RPUSHX key value

为已存在的列表添加值

## Set类型

在Redis中，我们可以将Set类型看作为没有排序的字符集合，和List类型一样，我们也可以在该类型的数据值上执行添加、删除或判断某一元素是否存在等操作。需要说明的是，这些操作的时间复杂度为O(1)，即常量时间内完成次操作。Set可包含的最大元素数量是4294967295。

和List类型不同的是，Set集合中不允许出现重复的元素。和List类型相比，Set类型在功能上还存在着一个非常重要的特性，即在服务器端完成多个Sets之间的聚合计算操作，如unions、intersections和differences。由于这些操作均在服务端完成，因此效率极高，而且也节省了大量的网络IO开销。

### 相关命令

1 SADD key member1 [member2]

向集合添加一个或多个成员

2 SCARD key

获取集合的成员数

3 SDIFF key1 [key2]

返回给定所有集合的差集

4 SDIFFSTORE destination key1 [key2]

返回给定所有集合的差集并存储在 destination 中

5 SINTER key1 [key2]

返回给定所有集合的交集

6 SINTERSTORE destination key1 [key2]

返回给定所有集合的交集并存储在 destination 中

7 SISMEMBER key member

判断 member 元素是否是集合 key 的成员

8 SMEMBERS key

返回集合中的所有成员

9 SMOVE source destination member

将 member 元素从 source 集合移动到 destination 集合

10 SPOP key

移除并返回集合中的一个随机元素

11 SRANDMEMBER key [count]

返回集合中一个或多个随机数

12 SREM key member1 [member2]

移除集合中一个或多个成员

13 SUNION key1 [key2]

返回所有给定集合的并集

14 SUNIONSTORE destination key1 [key2]

所有给定集合的并集存储在 destination 集合中

15 SSCAN key cursor [MATCH pattern] [COUNT count]

迭代集合中的元素

## Sorted Set类型

Sorted Set（有序集合）和Set类型极为相似，它们都是字符串的集合，都不允许重复的成员出现在一个Set中。它们之间的主要差别是Sorted Set中的每一个成员都会有一个分数(score)与之关联，Redis正是通过分数来为集合中的成员进行从小到大的排序。然而需要额外指出的是，尽管Sorted Set中的成员必须是唯一的，但是分数(score)却是可以重复的。

在Sorted Set中添加、删除或更新一个成员都是非常快速的操作，其时间复杂度为集合中成员数量的对数。由于Sorted Set中的成员在集合中的位置是有序的，因此，即便是访问位于集合中部的成员也仍然是非常高效的。事实上，Redis所具有的这一特征在很多其它类型的数据库中是很难实现的，换句话说，在该点上要想达到和Redis同样的高效，在其它数据库中进行建模是非常困难的。

### 相关命令

1 ZADD key score1 member1 [score2 member2]

向有序集合添加一个或多个成员，或者更新已存在成员的分数

2 ZCARD key

获取有序集合的成员数

3 ZCOUNT key min max

计算在有序集合中指定区间分数的成员数

4 ZINCRBY key increment member

有序集合中对指定成员的分数加上增量 increment

5 ZINTERSTORE destination numkeys key [key ...]

计算给定的一个或多个有序集的交集并将结果集存储在新的有序集合 key 中

6 ZLEXCOUNT key min max

在有序集合中计算指定字典区间内成员数量

7 ZRANGE key start stop [WITHSCORES]

通过索引区间返回有序集合成指定区间内的成员

8 ZRANGEBYLEX key min max [LIMIT offset count]

通过字典区间返回有序集合的成员

9 ZRANGEBYSCORE key min max [WITHSCORES] [LIMIT]

通过分数返回有序集合指定区间内的成员

10 ZRANK key member

返回有序集合中指定成员的索引

11 ZREM key member [member ...]

移除有序集合中的一个或多个成员

12 ZREMRANGEBYLEX key min max

移除有序集合中给定的字典区间的所有成员

13 ZREMRANGEBYRANK key start stop

移除有序集合中给定的排名区间的所有成员

14 ZREMRANGEBYSCORE key min max

移除有序集合中给定的分数区间的所有成员

15 ZREVRANGE key start stop [WITHSCORES]

返回有序集中指定区间内的成员，通过索引，分数从高到底

16 ZREVRANGEBYSCORE key max min [WITHSCORES]

返回有序集中指定分数区间内的成员，分数从高到低排序

17 ZREVRANK key member

返回有序集合中指定成员的排名，有序集成员按分数值递减(从大到小)排序

18 ZSCORE key member

返回有序集中，成员的分数值

19 ZUNIONSTORE destination numkeys key [key ...]

计算给定的一个或多个有序集的并集，并存储在新的 key 中

20 ZSCAN key cursor [MATCH pattern] [COUNT count]

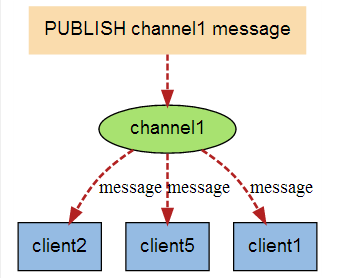
迭代有序集合中的元素（包括元素成员和元素分值）

# 发布订阅与消息队列

## 发布订阅模式

Redis的发布订阅模式包含两种角色，分别是发布者和订阅者。发布者(pub)发布消息，订阅者(sub)接收消息。发布者和订阅者类似与广播和听众的关系，广播可以选择一个频道发布消息，听众如果收听了这个频道，可以接受到广播发送的消息。

在发布订阅模式中，订阅者可以订阅一个或多个频道（channel），而发布者可以向指定的频道发送消息，所有订阅此频道的订阅者都会收到此消息。



在上图中，有一个channel1的频道和他的三个订阅者client2，client5，client1。当广播执行publish channel1 message命令，借助频道（channel1）发布消息（message）时，订阅频道channel1的三个订阅者client2，client5，client1都能够收到这个消息（message）。

客户端进入订阅模式后，可能接收到三种类型的消息。

1. 订阅频道时：第一个值是Subscribe，表示订阅成功；第二个值是订阅成功的频道名称；第三个值是当前客户端订阅的频道数。
2. 订阅模式中：第一个值是message，表示接收到发布者发布的消息；第二个值表示产生消息的频道名称；第三个值是消息内容。
3. 取消订阅：第一个值是unsubscribe。表示成功取消订阅某个频道；第二个值是对应的频道名称；第三个值是当前客户端订阅的频道数量，当此值为0时客户端会退出订阅状态。

发布订阅模式缺点：

1. 如果一个客户端订阅了频道，但自己读取消息的速度却不够快的话，那么不断积压的消息会使redis输出缓存区的体积越变越大，这可能使redis本身的执行速度变慢，甚至直接崩溃。
2. 数据可靠性问题，如果订阅方因意外断线，那么它会丢失所有在线期间发布者发布的消息。

### 相关命令

|  |  |
| --- | --- |
| SUBSCRIBE channel [channel ...] | 订阅给定的一个或多个频道的信息。 |
| PSUBSCRIBE pattern [pattern ...] | 订阅一个或多个符合给定模式的频道。 |
| PUBLISH channel message | 将信息发送到指定的频道。 |
| UNSUBSCRIBE [channel [channel ...]] | 退订给定的频道。 |
| PUNSUBSCRIBE [pattern [pattern ...]] | 退订所有给定模式的频道 |
| PUBSUB subcommand [argument [argument ...]] | 查看订阅与发布系统状态。 |

注意：

PSUBSCRIBE pattern [pattern ...] 订阅一个或多个符合给定模式的频道，后边跟的是一个表达式，可以用表达式匹配满足规则的频道

psubscribe channel.? 可以订阅channel.1，channel.0等频道

psubscribe channel.\* 可以订阅channel.1，channel.10等频道

### 应用场景

1. 最明显的用法就是构建实时消息系统，比如普通的即时聊天，群聊等功能
2. , 以门户网站为例, 当编辑更新了某推荐板块的内容后：CMS发布清除缓存的消息到channel (推送者推送消息)；门户网站的缓存系统通过channel收到清除缓存的消息 (订阅者收到消息)，更新了推荐板块的缓存

### 实例

打开一个客户端，执行subscribe channel命令，可以直接订阅指定名的频道（不需要创建频道）。

127.0.0.1:6379> psubscribe redisCha?  
 Reading messages... (press Ctrl-C to quit)  
 1) **"psubscribe"** 2) **"redisCha?"** 3) (integer) 1

再打开一个客户端，使用命令：publish channel message可以直接通过频道channel发布消息message

127.0.0.1:6379> publish redisChat **"Redis is a caching technique"** (integer) 1

在发布者通过频道redisChat发布消息后，订阅redisChat频道的订阅者可以接收到发布者发布的消息。

127.0.0.1:6379> subscribe redisChat  
 Reading messages... (press Ctrl-C to quit)  
 1) **"subscribe"** 2) **"redisChat"** 3) (integer) 1  
 1) **"message"** 2) **"redisChat"** 3) **"Redis is a caching technique"**

## 消息队列

Redis中的列表类型为实现队列提供了便捷，列表类型有LPUSH和RPOP命令。如果想要实现任务队列，只需要让生产者将任务使用LPUSH命令加入到某个键中，另一边让消费者不断地使用RPOP命令从该键中取出任务即可。

无限循环读取任务队列中的内容：

loop  
 **$task** = RPOP queue  
 **if** $task  
 #如果队列中有就执行  
 execute($task)  
 **else** #如果没有则等待1秒以免过于频繁地请求数据  
 wait 1 second

但是还有一些美中不足的地方，当任务队列中没有任务时消费者每秒都会调用一次RPOP命令查看是否有新任务。如果可以实现一旦有新任务加入任务队列就通知消费者就好了。使用BRPOP可以实现这样的需求。

BRPOP命令和RPOP命令相似，唯一区别就是当列表中没有元素时BRPOP命令会一直阻塞住连接，直到有新元素加入。

loop  
 #如果任务队列中没有新任务，BRPOP命令会一直堵塞，不会执行execute()  
 $task = BRPOP queue, 0  
 #返回值是一个数字，数组第二个值使我们需要的任务。  
 execute($task[1])

BRPOP命令接收两个参数，第一个是键名，第二个是超时时间，单位是秒。当超过此时间仍然没有获得新元素的话就会返回nil。超时时间是“0”表示不设置超时时间，即如果没有新元素加入队列就会永远阻塞下去。

示例，打开两个redis-cli:

实例A输入：

redis>BRPOP queue 0

实例B输入：

redis>LPUSH queue task

可以看到实例A返回：

1) **"queue"**2) **"task"**

同时发现queue中的元素已经被取走：

redis>LLEN queue  
(integer) 0

### 优先级队列

如果有两种任务同时存在，一种任务应该优先另一种任务处理，我们要如何使用队列呢？

为了实现这一目标，我们需要使用优先级队列，将不同的任务放在不同的任务队列中。BRPOP命令可以同时接收多个键，其完整的命令格式为BRPOP key [key ...] timeout，如BRPOP queue:1 queue:2 0.

意义是同时检测多个键，如果所有的键都没有元素则阻塞，如果其中有一个键有元素则会从该键中弹出元素。如果多个键都有元素则按照从左到右的顺序取第一个键中的一个元素。借此特性可以区分优先级队列。

比如通过命令：lpush queue:1 task1向队列queue:1中添加一个元素task1；通过命令：lpush queue:2 task2向队列queue:2中添加一个元素task2.

然后通过命令：brpop queue:1 queue:2 queue:3 0，则首先会获取队列一queue:1中的元素，然后才会获取队列二queue:2中的元素。

# 服务器和客户端

## Redis客户端

Redis 通过监听一个 TCP 端口或者 Unix socket 的方式来接收来自客户端的连接，当一个连接建立后，Redis 内部会进行以下一些操作：

1. 首先，客户端 socket 会被设置为非阻塞模式，因为 Redis 在网络事件处理上采用的是非阻塞多路复用模型。
2. 然后为这个 socket 设置 TCP\_NODELAY 属性，禁用 Nagle 算法
3. 然后创建一个可读的文件事件用于监听这个客户端 socket 的数据发送

常用Redis客户端命令：

1 CLIENT LIST 返回连接到 redis 服务的客户端列表  
2 CLIENT SETNAME 设置当前连接的名称  
3 CLIENT GETNAME 获取通过 CLIENT SETNAME 命令设置的服务名称  
4 CLIENT PAUSE 挂起客户端连接，指定挂起的时间以毫秒计  
5 CLIENT KILL 关闭客户端连接

## Redis服务器

Redis 服务器命令主要是用于管理 redis 服务，以下是常用的Redis服务器命令。

1 BGREWRITEAOF  
 异步执行一个 AOF（AppendOnly File） 文件重写操作  
 2 BGSAVE  
 在后台异步保存当前数据库的数据到磁盘  
 3 CLIENT KILL [ip:port] [ID client-id]  
 关闭客户端连接  
 4 CLIENT LIST  
 获取连接到服务器的客户端连接列表  
 5 CLIENT GETNAME  
 获取连接的名称  
 6 CLIENT PAUSE timeout  
 在指定时间内终止运行来自客户端的命令  
 7 CLIENT SETNAME connection-name  
 设置当前连接的名称  
 8 CLUSTER SLOTS  
 获取集群节点的映射数组  
 9 COMMAND  
 获取 Redis 命令详情数组  
 10 COMMAND COUNT  
 获取 Redis 命令总数  
 11 COMMAND GETKEYS  
 获取给定命令的所有键  
 12 TIME  
 返回当前服务器时间  
 13 COMMAND INFO command-name [command-name ...]  
 获取指定 Redis 命令描述的数组  
 14 CONFIG GET parameter  
 获取指定配置参数的值  
 15 CONFIG REWRITE  
 对启动 Redis 服务器时所指定的 redis.conf 配置文件进行改写  
 16 CONFIG SET parameter value  
 修改 redis 配置参数，无需重启  
 17 CONFIG RESETSTAT  
 重置 INFO 命令中的某些统计数据  
 18 DBSIZE  
 返回当前数据库的 key 的数量  
 19 DEBUG OBJECT key  
 获取 key 的调试信息  
 20 DEBUG SEGFAULT  
 让 Redis 服务崩溃  
 21 FLUSHALL  
 删除所有数据库的所有key  
 22 FLUSHDB  
 删除当前数据库的所有key  
 23 INFO [section]  
 获取 Redis 服务器的各种信息和统计数值  
 24 LASTSAVE  
 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  
 25 MONITOR  
 实时打印出 Redis 服务器接收到的命令，调试用  
 26 ROLE  
 返回主从实例所属的角色  
 27 SAVE  
 异步保存数据到硬盘  
 28 SHUTDOWN [NOSAVE] [SAVE]  
 异步保存数据到硬盘，并关闭服务器  
 29 SLAVEOF host port  
 将当前服务器转变为指定服务器的从属服务器(slave server)  
 30 SLOWLOG subcommand [argument]  
 管理 redis 的慢日志  
 31 SYNC  
 用于复制功能(replication)的内部命令

## Redis连接

Redis 连接命令主要是用于连接 redis 服务。以下实例演示了客户端如何通过密码验证连接到 redis 服务，并检测服务是否在运行：

redis 127.0.0.1:6379> AUTH **"password"**OK  
redis 127.0.0.1:6379> PING  
PONG

Redis连接常用命令：

1 AUTH password  
验证密码是否正确  
2 ECHO message  
打印字符串  
3 PING  
查看服务是否运行  
4 QUIT  
关闭当前连接  
5 SELECT index  
切换到指定的数据库

# 数据备份和恢复

## 数据备份

Redis提供了SAVE命令用于创建当前数据库的备份。

redis Save 命令基本语法如下：

redis 127.0.0.1:6379> SAVE  
OK

该命令将在 redis 安装目录中创建dump.rdb文件。

创建 redis 备份文件也可以使用命令 BGSAVE，该命令在后台执行

示例如下：

127.0.0.1:6379> BGSAVE  
Background saving started

## 数据恢复

如果需要恢复数据，只需将备份文件 (dump.rdb) 移动到 redis 安装目录并启动服务即可。获取 redis 目录可以使用 CONFIG 命令，如下所示：

redis 127.0.0.1:6379> CONFIG GET dir  
1) **"dir"**2) **"/usr/local/redis/bin"**

以上命令 CONFIG GET dir 输出的 redis 安装目录为 /usr/local/redis/bin

# 安全

Redis的安全设计是在"Redis运行在可信环境"这个前提下做出的，在生产环境运行时不允许外界直接连接到Redis服务器上，而应该通过应用程序进行中转。

运行在可信的环境中是保证Redis安全的最重要方法。

## 漏洞介绍

Redis 默认情况下，会绑定在 0.0.0.0:6379，这样将会将 Redis 服务暴露到公网上，如果在没有开启认证的情况下，可以导致任意用户在可以访问目标服务器的情况下未授权访问 Redis 以及读取 Redis 的数据。攻击者在未授权访问 Redis 的情况下可以利用 Redis 的相关方法，可以成功在 Redis 服务器上写入公钥，进而可以使用对应私钥直接登录目标服务器。

入侵特征：

1. Redis 可能执行过 FLUSHALL 方法，整个 Redis 数据库被清空
2. 在 Redis 数据库中新建了一个名为 crackit（网上流传的命令指令） 的键值对，内容为一个 SSH 公钥。
3. 在 /root/.ssh 文件夹下新建或者修改了 authorized\_keys 文件，内容为 Redis 生成的 db 文件，包含上述公钥

## 安全措施

Redis提供了以下几种方式用于提高Redis的安全性。Redis一般是不推荐使用登陆密码的，重命名Redis危险命令、指定连接Redis服务器的主机IP、使用单独的账号和目录启动Redis服务即可。

### 设置Redis登陆密码

为Redis服务器设置登陆密码在一定程度上可以防止别人随意连接服务器，修改服务器数据，保证数据的安全。Redis设置服务器登录密码有两种方式：

1. 在redis.windows.conf文件中设置requirepass选项：

requirepass hellocarl

1. 在客户端通过指令设置登陆密码：

CONFIG set requirepass "hellocarl"

由于Redis的性能极高，并且输入错误密码后Redis并不会设置延迟机制，比如错误多次后一定时间内不可再次尝试的等（考虑到Redis的单线程模型），所以攻击者可以通过穷举法破解Redis的密码（1秒内能够尝试十几万个密码），因此在设置时一定要选择复杂的密码。

使用命令行客户端配置密码，重启Redis后仍然会使用redis.conf配置文件中的密码。

### 危险命令重命名

Redis中提供了flushall命令，用于清除Redis缓存中的所有数据。一旦我们错误操作，或者有人故意操作，则数据数据集合直接被清空，这是非常危险的。为了解决这个问题，Redis提供了危险命令禁止或重命名的方法（类似于禁止快捷键或修改快捷键），在配置文件redis.windows.conf中设置：

rename-command FLUSHALL oyfekmjvmwxq5a9c8usofuo369x0it2k # 重命名FLUSHALL命令  
rename-command FLUSHALL "" # 禁用FLUSHALL命令

### 设置连接服务器的IP

Redis的默认配置会接受来自任何地址发送来的请求，即在任何一个拥有公网IP的服务器上启动Redis服务器，都可以被外界直接访问到。要更改这一设置，可以在配置文件中修改bind参数，如只允许本机或指定IP机器的应用连接Redis，可以将redis.windows.conf中的bind参数改成：

bind 127.0.0.1

### 在Linux下使用nobody启动Redis

在root账户下使用nobody用户启动程序xxx的方法：

su -m nobody -c xxx

nobody账户默认是无法登陆的，直接使用su -nobody会出现

This account is currently not available.

需要修改配置文件etc/passwd实现登陆

vim /etc/passwd  
找到  
nobody:x:99:99:Nobody:/:/sbin/nologin  
改为nobody:x:99:99:Nobody:/:/bin/bash  
后保存执行su -nobody

# 管道技术

Redis是一种基于客户端-服务端模型以及请求/响应协议的TCP服务。在Redis中一次请求过程如下：

1. 客户端向服务端发送一个查询请求，并监听Socket返回，通常是以阻塞模式，等待服务端响应；
2. 服务端处理命令，并将结果返回给客户端。

## 为什么使用管道技术

当客户端有大量命令需要请求服务器处理时，使用管道技术可以极大提升Redis处理效率。

在正常情况下，Redis客户端通过TCP报文发送一次请求，然后阻塞等待，服务器处理请求，然后以TCP报文形式返回响应结果。一次请求流程结束。

所以，每次请求都需要两个TCP报文。而我们知道，网络传输是有延迟的。因此，当Redis客户端发送大量请求时，由于网络延迟的原因，大部分的时间都浪费在了客户端阻塞上，这大大影响了Redis的高性能。

举例：

假如客户端和服务器之间的报文传输网络延迟为1.25s，那么1s之内网络只能传输8个报文，也就是1s内Redis的实际工作量是只能处理客户端发送的4次请求。这种情况下，即使Redis每秒能处理100个命令，而我们的客户端1s也只能发出4个请求。这显然浪费了Redis的处理能力，限制了Redis的性能。

因此，为了解决这种浪费，除了可以使用mget、mset命令一次处理多个键外，Redis还提供了对管道技术的支持。使用管道技术，可以在client打包多条命令一起发出，不需要等待单条命令的响应返回，而redis服务端也会处理完多条命令后将多条命令的处理结果打包到一起返回给客户端。这样，Redis虽然处理了多条命令，但实际上也只传输了两条报文，大大降低了网络延迟的影响。

通过pipeline方式当有大批量操作的时候。我们可以节省很多原本浪费在网络延迟上的时间。但需要注意的是用 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。因此，打包的命令越多，缓存消耗内存也越多。所以并是不是打包的命令越多越好。具体多少合适需要根据具体情况测试。

## 性能测试

下面是一个Jedis测试pipeline的测试：

**public class** PipelineTest {  
  
 **public static void** main(String[] args){  
 **int** count = 100000;  
 **long** start = System.*currentTimeMillis*();  
 *withoutPipeline*(count);  
 **long** end = System.*currentTimeMillis*();  
 System.***out***.println(end-start);  
 start = System.*currentTimeMillis*();  
 *usePipeline*(count);  
 end = System.*currentTimeMillis*();  
 System.***out***.println(end-start);  
 }  
  
 **public static void** withoutPipeline(**int** count){  
 Jedis jedis = **null**;  
 **try** {  
 jedis = **new** Jedis(**"127.0.0.1"**, 6379);  
 **for** (**int** i=0; i<count; i++){  
 jedis.incr(**"testKey1"**);  
 }  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }**finally** {  
 **if**(jedis != **null**){  
 jedis.disconnect();  
 }  
 }  
 }  
  
 **public static void** usePipeline(**int** count){  
 Jedis jedis = **null**;  
 **try** {  
 jedis = **new** Jedis(**"127.0.0.1"**, 6379);  
 Pipeline pipeline = jedis.pipelined();  
 **for** (**int** i=0; i<count; i++){  
 pipeline.incr(**"testKey1"**);  
 }  
 pipeline.sync();  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }**finally** {  
 **if** (jedis != **null**) {  
 jedis.disconnect();  
 }  
 }  
 }  
}

程序执行结果：

C:\Users\hand\AppData\Local\Temp\1511513536(1).png

由此可见，当客户端有大量请求时，采用pipeline方式可以极大提高服务响应效率。

# 持久化

Redis是一个支持持久化的内存数据库。如果不开启持久化功能，即单纯的把redis作为一个内存数据库，那么当Redis重启后，内存中的数据将全部丢失。这绝不是我们想要的。

Redis持久化功能能够将内存中的数据保存到磁盘中，当Redis重启时，再从磁盘中读取恢复数据。

Redis事务redis持久化存储支持两种方式：RDB（快照）和AOF。

RDB持久化是指在指定的时间间隔内将内存中的数据集快照写入磁盘

AOF持久化以日志的形式记录服务器所处理的每一个写、删除操作，查询操作不会记录。

官方建议两种方式同时使用，没有持久化的redis和memcache一样，相当于一个纯内存的数据库

## RDB(快照)

RDB(Redis Database)持久化是在指定的时间间隔内将内存中的数据集快照写入磁盘。

持久化步骤：

1. 在配置文件中会对RDB持久化的时间间隔进行配置。
2. 当需要持久化时，Redis会调用一个fork子进程将上一时间间隔内的数据集快照通过IO流写入一个临时性的dump.rdb文件。主进程不进行任何IO操作，继续对client的请求进行响应。
3. 当子进程完成对新 RDB 文件的写入时，Redis 用新 RDB 文件替换原来的 RDB 文件，并删除旧的 RDB 文件

快照是Redis默认的持久化方式。这种方式是将内存中数据以快照的方式写入到二进制文件中,默认的文件名为dump.rdb。可以通过配置设置自动做快照持久化的方式。我们可以配置redis在n秒内如果超过m个key被修改就自动做快照，下面是默认的快照保存配置：

save 900 1 #900秒内如果超过1个key被修改，则发起快照保存  
save 300 10 #300秒内容如超过10个key被修改，则发起快照保存  
save 60 10000

通过命令持久化：

client 也可以使用save或者bgsave命令通知redis做一次快照持久化。save操作是在主线程中保存快照的，由于redis是用一个主线程来处理所有 client的请求，这种方式会阻塞所有client请求。所以不推荐使用。另一点需要注意的是，每次快照持久化都是将内存数据完整写入到磁盘一次，并不是增量的只同步脏数据。如果数据量大的话，而且写操作比较多，必然会引起大量的磁盘io操作，可能会严重影响性能。

另外由于快照方式是在一定间隔时间做一次的，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。如果应用要求不能丢失任何修改的话，可以采用aof持久化方式。

RDB优点：

1. 使用单独子进程来进行持久化，主进程不会进行任何IO操作，保证了redis的高性能
2. 采用RDB方式时，数据持久化将只产生一个文件。这种文件非常适合用于进行备份： 比如说，你可以在最近的 24 小时内，每小时备份一次 RDB 文件，并且在每个月的每一天，也备份一个 RDB 文件。 这样的话，即使遇上问题，也可以随时将数据集还原到不同的版本。相比于AOF机制，如果数据集很大，RDB的启动效率会更高。
3. 对于灾难恢复而言，RDB是非常不错的选择。因为我们可以非常轻松的将一个单独的文件压缩后再转移到其它存储介质上

RDB缺点：

1、RDB是间隔一段时间进行持久化，如果持久化之间redis发生故障，会发生数据丢失。所以这种方式更适合数据要求不严谨的时候

## AOF

AOF是将执行过的指令以日志的形式记录下来，数据恢复时按照从前到后的顺序再将指令执行一遍，实现数据恢复。

AOF比快照方式有更好的持久化性，它支持以下三种持久化策略：

appendonly yes #启用aof持久化方式  
appendfsync always #每次收到写命令就立即强制写入磁盘，效率低，但是保证完全的持久化，不推荐使用  
appendfsync everysec #每秒钟强制写入磁盘一次，在性能和持久化方面做了很好的折中，推荐  
appendfsync no #完全依赖os，性能最好,持久化没保证

AOF文件重写：

Redis 可以在 AOF 文件体积变得过大时，自动地在后台对 AOF 进行重写： 重写后的新 AOF 文件包含了恢复当前数据集所需的最小命令集合。 整个重写操作是绝对安全的，因为 Redis 在创建新 AOF 文件的过程中，会继续将命令追加到现有的 AOF 文件里面，即使重写过程中发生停机，现有的 AOF 文件也不会丢失。 而一旦新 AOF 文件创建完毕，Redis 就会从旧 AOF 文件切换到新 AOF 文件，并开始对新 AOF 文件进行追加操作。AOF 文件有序地保存了对数据库执行的所有写入操作， 这些写入操作以 Redis 协议的格式保存， 因此 AOF 文件的内容非常容易被人读懂， 对文件进行分析（parse）也很轻松。 导出（export） AOF 文件也非常简单： 举个例子， 如果你不小心执行了 FLUSHALL 命令， 但只要 AOF 文件未被重写， 那么只要停止服务器， 移除 AOF 文件末尾的 FLUSHALL 命令， 并重启 Redis ， 就可以将数据集恢复到 FLUSHALL 执行之前的状态。

AOF优点：

1. 可以保持更高的数据完整性，如果设置追加file的时间是1s，即使redis发生故障，最多会丢失1s的数据；
2. AOF文件没被rewrite之前（文件过大时会对命令进行合并重写），可以删除其中的某些命令（比如误操作的flushall）
3. 由于该机制对日志文件的写入操作采用的是append模式，因此在写入过程中即使出现宕机现象，也不会破坏日志文件中已经存在的内容。即使本次操作只是写入了一半数据就出现了系统崩溃问题，在Redis下一次启动之前，我们也可以通过redis-check-aof工具来帮助我们解决数据一致性的问题。
4. 如果日志过大，Redis可以自动启用rewrite机制。即Redis以append模式不断的将修改数据写入到老的磁盘文件中，同时Redis还会创建一个新的文件用于记录此期间有哪些修改命令被执行。因此在进行rewrite切换时可以更好的保证数据安全性。

AOF缺点：

1. AOF文件比RDB文件大，且恢复速度慢
2. 随着写入命令的不断增加， AOF 文件的体积也会变得越来越大。举个例子， 如果你对一个计数器调用了 100 次 INCR ， 那么仅仅是为了保存这个计数器的当前值， AOF 文件就需要使用 100 条记录（entry）。然而在实际上， 只使用一条 SET 命令已经足以保存计数器的当前值了， 其余 99 条记录实际上都是多余的。
3. 根据同步策略的不同，AOF在运行效率上往往会慢于RDB。总之，每秒同步策略的效率是比较高的，同步禁用策略的效率和RDB一样高效

## RDB和AOF应用

一般情况下，推荐RDB和AOF两种持久化方式一起使用。如果你非常关心你的数据,但仍然可以承受数分钟以内的数据丢失， 那么你可以只使用 RDB 持久化。有很多用户都只使用 AOF 持久化， 但我们并不推荐这种方式： 因为定时生成 RDB 快照（snapshot）非常便于进行数据库备份， 并且 RDB 恢复数据集的速度也要比 AOF 恢复的速度要快，

# 事务

Redis事务提供了一种“将多个命令打包，然后一次性、按顺序地执行”的机制，它提供两个保证：

• 事务是一个单独的隔离操作：事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断

• 事务是一个原子操作：事务中的命令要么全部被执行，要么全部都不执行。Redis事务没有回滚操作，即使中间有一个命令发生异常，接下来的命令也会执行。

Redis事务从开始到执行会经历以下三个阶段：

开启事务 🡪 命令入队 🡪 执行事务

MULTI，EXEC，DISCARD，WATCH 四个命令是 redis 事务的四个基础命令。其中：

MULTI，告诉 redis 服务器开启一个事务。注意，只是开启，而不是执行

EXEC，告诉 redis 开始执行事务

DISCARD，告诉 redis 取消事务

WATCH，监视某一个键值对，它的作用是在事务执行之前如果监视的键值被修改，事务会被取消。

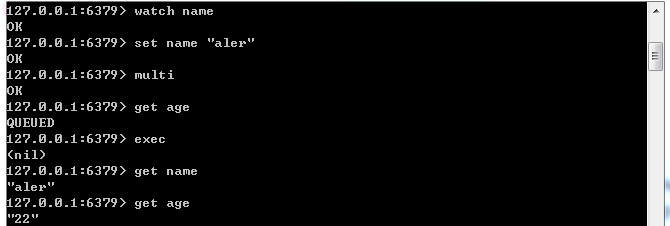
示例：

## 相关命令

|  |  |
| --- | --- |
| 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。 |

### Watch命令

watch命令可以监控一个或多个键。在watch命令和事务执行之前，也就是watch和exec命令之间，如果watch监控的键被修改或删除了，则事务就不会执行，即使事务中不存在涉及watch监控的键的命令。



## 事务内部错误

在一个事务的运行期间，可能会遇到两种类型的命令错误：

1. 事务在执行EXEC 之前，入队的命令可能会出错。比如说，命令可能会产生语法错误（参数数量错误，参数名错误，等等），或者其他更严重的错误，比如内存不足（如果服务器使用 maxmemory 设置了最大内存限制的话）
2. 命令可能在EXEC 调用之后失败。举个例子，事务中的命令可能处理了错误类型的键，比如将列表命令用在了字符串键上面，诸如此类

对于发生在EXEC 执行之前的错误，客户端以前的做法是检查命令入队所得的返回值：如果命令入队时返回 QUEUED ，那么入队成功；否则，就是入队失败。如果有命令在入队时失败，那么大部分客户端都会停止并取消这个事务

不过，从 Redis 2.6.5 开始，服务器会对命令入队失败的情况进行记录，并在客户端调用EXEC 命令时，拒绝执行并自动放弃这个事务。

## 为什么不支持回滚

如果你有使用关系式数据库的经验，那么“Redis 在事务失败时不进行回滚，而是继续执行余下的命令”这种做法可能会让你觉得有点奇怪

以下是这种做法的优点：

• Redis 命令只会因为错误的语法而失败（并且这些问题不能在入队时发现），或是命令用在了错误类型的键上面：这也就是说，从实用性的角度来说，失败的命令是由编程错误造成的，而这些错误应该在开发的过程中被发现，而不应该出现在生产环境中

• 因为不需要对回滚进行支持，所以 Redis 的内部可以保持简单且快速有种观点认为 Redis 处理事务的做法会产生 bug ，然而需要注意的是，在通常情况下，回滚并不能解决编

程错误带来的问题。举个例子，如果你本来想通过INCR 命令将键的值加上 1 ，却不小心加上了 2 ，又或者对错误类型的键执行了INCR ，回滚是没有办法处理这些情况的

鉴于没有任何机制能避免程序员自己造成的错误，并且这类错误通常不会在生产环境中出现，所以 Redis 选

择了更简单、更快速的无回滚方式来处理事务

# 缓存机制

# Redis锁机制

一般情况下，锁机制的流程是：

1. 程序首先获取锁，得到对数据进行排他性访问的能力。也就是当前线程获取数据锁时，其他线程不能操作数据。
2. 获取到锁之后，对数据执行一系列的操作。
3. 数据操作完成后，释放锁。

乐观锁和悲观锁：

1. 乐观锁：在关系型数据库中，乐观并发控制（又称乐观锁，OCC）是一种并发控制的方法。使用乐观锁时，在每次获取数据打算操作的时候，它都默认别人不会修改这条数据，所以不会上锁。但是在自己数据更新之后要提交的时候，它就会判断一下在它获取数据到更新提交整个过程内，是不是有其他线程对这条数据进行了修改（可以在表中加入一个版本号来实现）。如果数据被修改了，它就会回滚。
2. 悲观锁：在关系型数据库中，悲观并发控制（又称悲观锁，PCC）是一种并发控制的方法。悲观锁，顾名思义，就是比较悲观。每次获取数据后，都觉得这条数据可能会被其他线程修改。所以在每次获取数据后，都会对这个数据上锁。这时候其他线程就不能获取到这条数据，自然更不可能修改数据。只有当当前线程完成数据操作并提交后，然后释放锁，这时其他线程才可以访问数据。

举例：

乐观锁，就类似生活中比较乐观的人。总觉得不会有人偷自己的东西，比如，小明经常把钱放在抽屉里，然后也不上锁。每次花钱时呢，打开抽屉拿出钱就可以花了，特别方便，也不用上锁开锁的。但是呢，每次从抽屉取钱用的时候，小明都会检查下自己还有多少钱。如果在他把钱放进抽屉到他取钱花的这个期间，他的钱被人偷了，这时候肯定就不爽了啊，然后就报警选择啊。

悲观锁呢，就像生活中比较悲观的人，有些人总觉得生活中充满危险。比如，小亮总觉得自己有人偷自己的东西。因此，他把自己的东西全部锁紧保险柜里。这时候，有人就好奇小亮的保险柜里有啥呢？可是他上锁了，你看不到啊。所以你只能等小亮自己开锁之后，才可以看看。

因此，乐观锁适用于资源竞争、数据冲突不激烈的情况下。乐观锁会先操作数据，在更新的时候才会检查数据有没有被修改，有修改会回滚事务。如果数据竞争激烈，大量事务回滚会严重影响系统性能。

悲观锁适用于资源竞争激烈的情况下。悲观锁为安全处理数据提供了保证，但是数据加锁和释放锁的操作影响了数据库效率和并发性，并且可能引发死锁。在只读事务中，不会修改数据，因此不会产生冲突，没必要加锁。

Redis提供了Watch命令，用于代替对数据加锁。

举例：

如果客户端1对数据A执行了watch命令。然后当客户端1和客户端2一起对数据A进行修改，此时，watch命令并不会阻止客户端2对数据A的修改，它只会告诉客户端1，数据A被修改了。起到了一个监控作用，类似用乐观锁。

我们没有直接使用操作系统级别的锁、编程语言级别的锁，或者其他各式各样的锁，而是选择花费大量时间去使用Redis构建锁，这其中一个原因和范围有关：为了对Redis存储的数据进行排他性访问看，客户端需要访问一个锁，且这个锁需要定义在一个可以被所有客户端看见的范围之内，而这个范围就是Redis本身，因此我们需要把锁构建在Redis里面。

## 分布式锁

分布式锁虽然也有类似的“先获取锁，然后操作数据，最后释放锁”流程。但分布式锁既不是给同一个进程中的多个线程使用的，也不是给同一台机器上的多个进程使用，而是由不同的机器上的不同Redis客户端进行获取和释放的。针对的是使用系统服务的不同机器。

使用Redis构建分布式锁的原因：

1. Redis有很高的性能
2. Redis命令对分布式锁的支持比较好，实现方式比较简单

构建Redis锁需要考虑的场景：

1. 持有锁的进程因为操作时间太长导致锁被释放，但进程本身并不知道这点。甚至当进程执行完后，还会错误释放掉其他进程持有的锁。
2. 一个持有锁并且打算执行长时间操作的进程崩溃掉，但是其他想要获取锁的进程并不知道哪个进程持有锁，也不知道持有锁的进程已经崩溃，仍会傻傻的浪费时间等待锁被释放。
3. 在一个进程持有的锁过期之后，其他多个进程同时尝试获取锁，并且都获得了锁。
4. 上面第一种情况和第三种情况同时出现，导致多个进程获得了锁，并且每个进程都以为自己是唯一一个获得锁的进程。

Redis锁机制：

构建Redis锁常用命令:

1. SETNX命令

SETNX命令天生就适合用来实现锁的获取功能。SETNX命令会为键设值，但只有当键不存在时，才会设置;如果键已经有对应值，则什么也不做。

1. EXPIRE命令

EXPIRE命令可以为一个键设置超时时间。当超过设置的时间之后，这个键和它对应的值会自动被删除。

1. DEL命令

DEL命令用于删除键对应的值。

https://www.cnblogs.com/liuyang0/p/6744076.html

## 计数信号量

计数信号量是一种锁，它可以让用户限制一项资源最多能够同时被多少个进程访问，通常用于限定能够同时使用的资源数量。

计数信号量的锁并没有一般的锁那么常用，但它有独特的应用场景。当我们需要让多个客户端同时访问相同的信息时，计数信号量就是完成这项任务的最佳工具。

# 主从复制

Redis的主从复制就是一个主服务器可以配置一个或多个从服务器(可以是级联结构)，建立多个和主服务器完全一样的数据库环境。

## 为什么使用主从复制？

1. 主服务器一般是实时的业务数据库。如果主服务器发生故障导致宕机，不仅服务崩溃了，还可能导致数据丢失。当使用主从复制时，即使主服务器发生故障，Redis也可以切换到从服务器继续工作。
2. Redis虽然性能很高，但也会产生读写压力特别大的情况。从服务器可以在不影响主服务器性能的前提下进行数据备份、统计等工作。相当于从服务器分担了主服务器的部分工作，间接提升了主服务器的性能。

redis在2.4版本后加入了sentinel功能，主要功能是在主机宕机时自动选举出一个slave，并且将其转换为master，保证业务的正常运行。

## Redis主从复制特性

1. 一个主服务器可以配置多个从服务器，可以是级联结构；
2. 从服务器除了可以连接到主服务器外，也可以多个从服务器连接到一个从服务器；
3. redis复制在master这一端是非阻塞的，也就是说在和 slave 同步数据的时候，master 仍然可以执行客户端的操作命令而不受其影响；
4. redis复制在slave这一端也是非阻塞的，在redis配置文件里面有 slave-serve-stale-data 这一项，如果它为yes，当slave执行同步时，它会使用老版本的数据来处理查询请求。如果为no，当slave执行同步时，它会返回给客户端一个错误信息。同步完成后，slave会删除老数据，加载新数据，在这个阶段，它会阻止连接请求。
5. 主从复制可用来对Redis进行性能扩展，可以使用多个从服务器来处理只读的请求（比如，繁重的排序操作可以放到从服务器去做），也可以简单的用来做数据冗余；
6. 使用主从复制可以免除主服务器把数据写入磁盘的消耗，在redis的配置文件中可以把master中的save配置项全部注释掉，不让它进行保存，然后配置slave，让slave保存。虽然有这个特性，但好像一般不用。

## 全量同步和增量同步

Redis主从复制在初始化阶段，从服务器slave需要将master中的数据都复制一份，也就是同步。这里的数据同步分为全量同步和增量同步。

Redis的主从同步策略是：在主从刚刚连接的时候，进行全量同步；全量同步结束后，进行增量同步。当然，如果有需要，slave 在任何时候都可以发起全量同步。redis 策略是，无论如何，首先会尝试进行增量同步，如果不成功，再尝试进行全量同步。

### 全量同步

Redis全量复制一般发生在Slave初始化阶段，这时Slave需要将Master上的所有数据都复制一份。

全量同步步骤如下：

1. 从服务器连接主服务器，发送sync命令；
2. 主服务器接收到SYNC命名后，开始执行BGSAVE命令生成RDB文件并使用缓冲区记录此后执行的所有写命令；
3. 主服务器BGSAVE执行完后，向所有从服务器发送快照文件，并在发送期间继续记录被执行的写命令；
4. 从服务器收到快照文件后丢弃所有旧数据，载入收到的快照；
5. 主服务器快照发送完毕后开始向从服务器发送缓冲区中的写命令；
6. 从服务器完成对快照的载入，开始接收命令请求，并执行来自主服务器缓冲区的写命令

完成上面几个步骤后，从服务器数据初始化流程结束，之后，从服务器就可以接收来自用户的读请求。

### 增量同步

Redis增量复制是指在Slave完成初始化后，主服务器正常工作时，会将写操作命令存入缓存区。然后当缓存区的写命令保存一定数量后，将缓存区的写操作同步到从服务器的过程。

增量复制的过程主要是主服务器每执行一个写命令就会向从服务器发送相同的写命令，从服务器接收并执行收到的写命令。

## 集群管理工具Redis-sentinel

Redis-sentinel概念

Redis-sentinel是Redis实例的监控管理、通知和实例失效备援服务，是Redis集群的管理工具。在一般的分布式中心节点数据库中，Redis-sentinel的作用是中心节点的工作，监控各个其他节点的工作情况并且进行故障恢复，来提高集群的高可用性

主要职责：

1. 监测：监控主从服务器等各个节点的工作情况；
2. 通知：当监控到某个master故障后，sentinel可以通知其他监控该master的sentinel；
3. 故障自动恢复：当主服务器发生故障时，选择一个从服务器代替主服务器工作，保证服务的正常进行。

Redis-sentinel的工作过程类似与一个班级的管理。比如一个班级有50个学生，然后有一个班主任，三个老师,。50个学生呢，分成5个小组，每个小组都有一个组长。这里的班主任和老师就相当于Redis-sentinel，组长相当于主服务器master，普通学生相当于从服务器slave。班主任和老师呢就负责监测着整个班级。然后有一天呢，第一组的组长A生病了，请了一个长假。也就是有一个master故障了。然后呢，班级工作要正常进行啊。所以呢，班主任就重新认命了第一组的B为新的组长，代替A工作。

### Sentinel工作机制

1、通过master获取master的sentinels和slaves

现在有一个Sentinel实例sentinelA，它首先会读取配置文件，配置文件中有一个master的ip和port列表，同时会指定连接失败超时的时间和是否可以故障恢复等参数信息。这样，sentinelA就获取了它要监控的master列表。

sentinelA监控master是通过发布订阅（pub/sub）机制实现的。比如sentinelA要监控master1，sentinelA就会订阅master1的‘sentinel：hello’频道，并且在这个频道publish自己的host信息。这样，所有监控master1的sentinel都知道了彼此的存在。SentinelA就得到了master1的sentinels列表，sentinelA还可以通过info命令获取master1的slave列表。如果连接超时或者失败，就会进入故障处理的流程。

2、故障恢复准备

Redis-sentinel为master的故障定义了两种状态，O\_DOWN和S\_DOWN。当sentinelA连接master1失败后，sentinelA就认定master1处于S\_DOWN状态。然后sentinelA就开始通过SENTINEL IS-MASTER-DOWN-BY-ADDR命令，获取master1的sentinels列表中的其他sentinel对master1的认定情况。如果有超过quorum(配置文件中指定)个sentinel认定master 1已经DOWN，那么就认定master 1的状态为O\_DOWN（相当于一种投票机制）。

sentinel A必须有master 1的can failover权限，也是配置文件指定的。接着，sentinel A需要得知master 1的leader sentinel，由leader来进行master 1的错误恢复。

master 1的leader sentinel由选举产生，每个监控master 1并且具有错误恢复权限的sentinel都需要进行一次叫做subjective leader的判断，也就是sentinel自己认定的leader。然后sentinel之间通过SENTINEL IS-MASTER-DOWN-BY-ADDR命令交流leader的认定情况，最终得到一个共识。

3、故障处理

sentinel A作为master 1的leader，会选取master 1的一个slave作为新的master。slave的选取是根据判断DNS情况的优先级来得到的，优先级相同通过runid的排序得到，但目前优先级设定还没实现，所以直接获取runid排序得到slave1。然后发送命令slaveof no one将 slave1的状态slave设为master。然后，当其他sentinel观察到该slave成为master后,就知道错误处理程序启动了。sentinel A将slaveof new-slave-ip-port命令发送给其他slave，其他slave重新设置自己master。当所有slave都配置完后，sentinel A从监测的masters列表中删除故障master，并通知其他sentinels。

## 注意点

场景1：当主服务器master不进行数据持久化时，主从复制时数据安全性问题？

Redis具有自动重启系统，当主服务器发生故障崩溃时，会自动重启一个新的进程，重启后的主服务器中只有一个空的数据集。在这个过程中，如果我们设置了Redis的主服务器不支持持久化，那么此时服务器中的数据是非常危险的。

比如主服务器可能在故障发生后很短的时间内就完成了重启，以至于Sentinel都没有检测到这次事故，自然也就没有让一个从服务器slave代替主服务器。但是主服务器确实又重启了，所以从服务器就会开始全量复制master中的数据，但这时候其实master中的数据集是空的。所以，好玩的事情发生了，从服务器中的数据集也变成空集了，也就是数据丢失了。

因此，在进行主从复制设置时，强烈建议在主服务器上开启持久化！如果不能这么做时，比如考虑到延迟的问题，应该将实例配置为禁止自动重启。

## 相关配置

## 测试

### 场景1：主从复制

1. 要进行主从复制测试，首先要启动2个或以上服务器实例，然后分别启动一个客户端连接它们。
2. 在Windows系统中启动多个Redis服务器，分别监听不同的端口。
3. 在Redis安装目录中找到redis.windows-service.conf文件，然后复制一份，并改名为redis.windows-service-6380.conf；
4. 修改复制文件内容：一是修改端口号port 6379为port 6381；二是修改logfile "server\_log.txt"为logfile "server\_6381\_log.txt"，如果文件中的syslog-enabled为no则不需要修改日志文件；
5. 在Redis安装目录下打开命令行窗口，执行redis-server redis.windows-service-6381.conf命令，此时命令行窗口中的光标会一直闪烁，为了判断是否成功，我们可以到Redis目录中查看b）中配置的日志文件是否生成；
6. 关闭c）中打开的命令行窗口，然后重新打开一个新的窗口。执行redis-server --service-install redis.windows-service-6381.conf --service-name redis6381 --port 6381命令，然后再执行redis-server --service-start --service-name redis6381命令，启动Redis服务器B，监听6381端口；
7. 双击Redis安装目录下的redis-server.exe文件，运行默认的Redis服务器A，监听6379端口；
8. 启动两个Redis客户端，分别监听6379和6381端口。
9. 在Redis安装目录下打开命令行窗口，执行redis-cli –p 6379命令，开启一个连接6379端口的客户端A；
10. 在Redis安装目录下打开命令行窗口，执行redis-cli –p 6381命令，开启一个连接6381端口的客户端B；
11. 验证主从复制：
12. 在客户端A中执行set replication “none”命令设置一个字符串变量，然后在客户端A和B中分别执行get replication命令，可以发现客户端A中返回了none，而客户端B什么也没有。这说明客户端A向服务器A中写入的数据只存入了服务器A中；
13. 在客户端B中执行slaveof 127.0.0.1 6379命令，将服务器B设置为服务器A的从服务器，然后再执行get replication命令，结果返回了none，说明服务器B已经复制了服务器A中的数据集；

### 场景2：故障切换

# 分区

## 什么是分区？

Redis分区，简单来说就是将数据分布到不同的Redis实例中，因此每个Redis实例存储的内容仅仅是总存储内容的一个子集。分区不是Redis独有的，几乎所有的数据库存储系统都会涉及到分区这个概念。

## 为什么要进行分区？

1. 提升性能：单机Redis的网络I/O能力和计算能力是有限的，充分利用多台机器的计算能力和网络带宽，有助于提升Redis总体的服务能力；
2. 存储的水平扩展：单台机器的存储容量是有限的，通过分区我们可以将数据分散到多台机器上，使Redis的存储能力横向扩展。

总之，分区可以帮助我们突破单台机器硬件资源的限制，比如计算能力、存储空间、网络带宽等。

## 分区实现方案

分区可以在程序的不同层次上实现。

1. 客户端分区：就是指在客户端就已经决定好了数据将要被存储到哪个Redis节点或者打算从哪个Redis节点读取数据。大多数客户端都实现了客户端分区。
2. 代理分区：意味着客户端请求会被发送给代理，然后由代理决定将数据写入哪个节点或者从哪个节点读数据，代理根据分区规则决定请求哪些Redis实例，并将Redis的响应结果返回给客户端。
3. 查询路由分区：就是在客户端随机请求任意一个Redis实例，然后由这个Redis实例将请求转发给正确的Redis节点。Redis Cluster实现了一种混合形式的查询路由，但它并不是由Redis实例将客户端请求转发给Redis节点，而是在客户端的帮助下直接重定向（redirected）正确的Redis节点。

## 分区的不足

1. 涉及多个key的操作通常是不被支持的。例如你不能对两个set求交集，因为它们可能被映射到不同的Redis实例上。
2. 涉及多个key的Redis事务不能使用。
3. 分区的最小粒度是键。因此，即使有一个键对应的数据集非常大，这个数据集也不能映射到不同的Redis实例上。
4. 当使用分区时，数据处理比较复杂。比如我们需要处理多个rdb/aof文件，将分布在不同实例的文件聚集到一起进行备份。
5. 增加或删除容量比较复杂。Redis集群在运行时增加或者删除Redis节点时，对数据再平衡的过程对用户几乎是透明的。但是如果使用的是客户端或代理分区，是不支持上述操作的。为了解决这个问题，Redis提供了presharding技术。

## 预分区Pre-Sharding

Redis虽然经常被用作缓存，但它也对数据持久化提供了支持。对于Redis来说，无论数据持久化存储还是缓存，数据分区在概念上是一样的。但是，在使用数据分区时，如果我们把Redis当做数据持久化存储来使用，就必须注意一个问题：就是一个Key无论被更新多少次，它都必须一直被映射到同一个Redis实例中；如果我们把Redis当做缓存使用，对于某个Key，如果它对应的实例不能用了，我们还可以把它映射到其它实例中

由上可知，除非我们只使用Redis中的缓存，否则我们想要动态增加或删除Redis节点是非常麻烦的。

一般情况下，随着时间的推移，数据存储需求总会发生变化。比如一款软件原来只设置了10个Redis节点，但是有一天它突然特别流行，10个节点的数据存储空间不够了或者计算能力不行了，那就要增加Redis节点。

我们都知道Redis是轻量级服务，每个实例仅仅占用1M，因此，在最开始的时候，我们可以在同一台机器上多启动一些Redis实例，以解决上面这种动态增加Redis节点的需求。这就是预分区。

比如：我们可以在同一台机器上开启多个Redis实例，然后从中选择一些实例，比如32或64个实例来作为我们的工作集群。当一台物理机器存储不够的时候，我们可以将一半的实例移动到我们的第二台物理机上，依次类对，我们可以保证集群中Redis的实例数不变情况下，达到扩充机器的目的。

那么怎么移动Redis实例呢？当需要将Redis实例移动到独立的机器上的时候，我们可以通过下面步骤实现：

1. 在新的物理机上启动一个新的Redis实例；
2. 将新的物理机作为原来那台机器的slave机；
3. 停止客户端；
4. 对于将要被移动的那些Redis实例，更新它们的IP地址；
5. 在slave机器执行SLAVEOF ON ONE命令；
6. 使用新的IP启动Redis客户端；
7. 关闭原服务器中不再使用的那些Redis实例。

## 分区类型

### 范围分区

最简单的分区方式是按范围分区，就是将一定范围的对象映射到特定的Redis实例中。

比如，ID从0到10000的用户会保存到实例R0，ID从10001到 20000的用户会保存到R1，以此类推。

这种方式是可行的，并且在实际中也经常使用。但是一个致命的不足就是：需要管理一个区间范围到实例的映射表，而且是每种对象都需要一张范围到实例的映射表来管理，比如用户对象，前10000个对象关联实例R0……订单对象，前10000张订单关联实例R0……甚至有时候我们需要存数的数据key不能按照范围划分，比如key是uuid

### 哈希分区

哈希分区跟范围分区相比一个明显的优点是哈希分区适合任何形式的key，而不像范围分区一样需要key的形式为object\_name:<id>。哈希分区像下面的描述一样简单：

1. 用一个hash函数将key转换为一个数字，比如使用crc32 hash函数。对key foobar执行crc32(foobar)会输出类似93024922的整数。
2. 对这个整数取模，比如我们有4个Redis实例，就将其转化为0-3之间的数字，然后将这个整数映射到4个Redis实例中的一个。93024922 % 4 = 2，就是说key foobar应该被存到R2实例中。

# redis集群

<http://blog.csdn.net/gongpulin/article/details/54178493>

<http://www.cnblogs.com/hjwublog/p/5681700.html#_label0>

<http://blog.csdn.net/javaloveiphone/article/details/52352894>

<http://www.sohu.com/a/79200151_354963>

简介

Redis集群是3.0之后才引入的。在3.0之前，Redis使用哨兵(Sentinel)机制来监控各个节点之间的状态。

Redis集群是一组能进行数据共享的Redis实例(服务或者节点)。集群可以使用的功能是普通单击Redis所能使用功能的子集。

Redis集群通常具有高可用性、水平扩展性、分布式和容错性等特性。

## Redis Cluster简介

redis cluster是redisr推出的集群版，对应版本是 redis 3.0版本。redis cluster后面一系列源码学习都是针对该版本的。

Redis cluster(集群)是分布式（distributed）的 Redis 实现，具有一定的容错性（fault-tolerant）和线性可扩展性（linear scalability）。

主要功能：

1. 可线性扩展到16384个节点（cluster使用哈希方式将数据分布到16384个槽，redis节点以槽为单位负责存储数据） ；
2. 通过cluster 客户端可使数据自动路由到所负责节点 ；
3. 实现了多个节点间的数据共享（主从节点数据共享；所有节点共享配置数据）；
4. 可支持动态增加或删除节点；
5. 可保证某些节点无法提供服务时不影响整个集群的操作；
6. 不保证数据的强一致性，但在不出现主从切换的情况下，从用户角度是可以保证数据强一致性
7. 可支持动态调整数据分布，从而实现负载匀衡
8. 可支持人工动态调整主从关系

命令：

1. 支持Redis所有处理单个数据库键的命令；
2. 不支持对多个数据库键的操作，比如MSET、SUNION；
3. 不能使用 SELECT 命令，集群只使用默认的0号数据库；
4. 增加cluster相关命令，用于配置和运维redis cluster。

## CAP原则

CAP原则是NoSql数据库的基石，在互联网界有着广泛的知名度，又被称为布鲁尔定理。

它的核心是：一个分布式系统不可能同时很好的满足一致性，可用性和分区容错性这三个需求，最多只能同时较好的满足两个。

一致性（Consistency)：

概念：

同一个数据在集群中的所有节点，同一时刻是否都是同样的值。

说明：

对于一致性，可以分为从客户端和服务端两个不同的视角。从客户端来看，一致性主要指的是多并发访问时更新过的数据如何获取的问题。从服务端来看，则是更新如何复制分布到整个系统，以保证数据最终一致。一致性是因为有并发读写才有的问题，因此在理解一致性的问题时，一定要注意结合考虑并发读写的场景。从客户端角度，多进程并发访问时，更新过的数据在不同进程如何获取的不同策略，决定了不同的一致性。对于关系型数据库，要求更新过的数据能被后续的访问都能看到，这是强一致性。如果能容忍后续的部分或者全部访问不到，则是弱一致性。如果经过一段时间后要求能访问到更新后的数据，则是最终一致性。

可用性（Availability）：

概念：

集群中一部分节点故障后，集群整体是否还能处理客户端的更新请求。

说明：

对于一个可用性的分布式系统，每一个工作正常的节点必须对每一个请求作出响应。好的可用性主要是指系统能够很好的为用户服务，不出现用户操作失败或者访问超时等用户体验不好的情况。

分区容忍性（Partition tolerance）：

概念：

是否允许数据的分区，分区的意思是指是否允许集群中的节点之间无法通信。

说明：

分区容错性和扩展性紧密相关。在分布式应用中，某些原因会导致系统无法正常运转。比如现在的分布式系统中有某一个或者几个机器宕掉了，其他剩下的机器还能够正常运转满足系统需求，或者是机器之间有网络异常，将分布式系统分隔未独立的几个部分，各个部分还能维持分布式系统的运作，这样就具有良好的分区容错性。

总结：

由于分布式系统，我们无法不选择P，因为由于网络的不可靠性，必定会导致两个机器节点之间无法进行网络通信，从而导致数据无法同步。这样的现实导致我们必须选择P。然后，在这个前提下，我们要么选择A，要么选择C。

原因如下：

当两台机器节点之间无法通信时（即选择了P），如果我们继续允许客户端写入数据到其中一台，那就必然导致数据不一致，因为这个数据无法同步到另一台机器。（这种做法就是选择了A，牺牲了C）

当两台机器节点之间无法通信时（即选择了P），如果我们不允许客户端写入数据到任何一台，那数据虽然是一致的，但系统就不可用了。（这种做法就是选择了C，牺牲了A）

# Java操作Redis

Java如果要操作非关系型内存数据库Redis，必须在机器上安装Redis的Java驱动程序Jedis。同时，如果要使用Redis的连接池或者在Spring中使用Redis，则必须在pom.xml文件中引入下面依赖：

Spring与Redis的集成

<**dependency**>  
 <**groupId**>org.springframework.data</**groupId**>  
 <**artifactId**>spring-data-redis</**artifactId**>  
 <**version**>1.8.1.RELEASE</**version**>  
</**dependency**>

Redis连接池  
<**dependency**>  
 <**groupId**>org.apache.commons</**groupId**>  
 <**artifactId**>commons-pool2</**artifactId**>  
 <**version**>2.4.2</**version**>  
</**dependency**>

Redis的Java驱动程序  
<**dependency**>  
 <**groupId**>redis.clients</**groupId**>  
 <**artifactId**>jedis</**artifactId**>  
 <**version**>2.9.0</**version**>  
</**dependency**>

在使用Java代码操作Redis时，首先需要获取Redis服务的连接：

**Jedis jedis** = **new** Jedis(**"127.0.0.1"**, 6379);  
**jedis**.auth(**""**); //auth方法中传入Redis服务器的用户密码

使用Java操作Redis字符串：

**jedis**.set(**"name"**, **"Someone"**);  
**logger**.error(**"姓名： "**+**jedis**.get(**"name"**));  
  
**jedis**.append(**"name"**, **" hurt me must"**);  
**logger**.error(**"姓名： "**+**jedis**.get(**"name"**));  
  
**jedis**.del(**"name"**);  
**logger**.error(**"姓名： "**+**jedis**.get(**"name"**));  
  
**jedis**.mset(**"name"**,**"Allin"**, **"age"**, **"22"**, **"email"**, **"weisp@vanke.com"**);  
**jedis**.mget(**"name"**, **"age"**, **"email"**).forEach(**logger**::info);

使用Java操作Redis的哈希列表：

Map<String, String> param = **new** HashedMap();  
param.put(**"name"**, **"Allin"**);  
param.put(**"age"**, **"22"**);  
param.put(**"email"**, **"weisp@vanke.com"**);  
  
**jedis**.hmset(**"user"**, param);  
**jedis**.hmget(**"user"**, **"name"**, **"age"**, **"email"**).forEach(**logger**::info);  
  
**jedis**.hdel(**"user"**, **"email"**);  
**jedis**.hmget(**"user"**, **"name"**, **"age"**, **"email"**).forEach(**logger**::info);  
  
**logger**.info(**"包含与否： "**+**jedis**.hexists(**"user"**, **"email"**));  
**jedis**.hkeys(**"user"**).forEach(**logger**::info);  
**jedis**.hvals(**"user"**).forEach(**logger**::info);

使用Java操作Redis的列表：

**jedis**.del(**"SSM"**);  
**jedis**.lpush(**"SSM"**, **"Spring"**);  
**jedis**.lpush(**"SSM"**, **"Spring MVC"**);  
**jedis**.lpush(**"SSM"**, **"Mybatis"**);  
  
**jedis**.lrange(**"SSM"**, 0, -3).forEach(**logger**::info);  
**jedis**.del(**"SSM"**);  
**jedis**.rpush(**"SSM"**, **"Spring"**);  
**jedis**.rpush(**"SSM"**, **"Spring MVC"**);  
**jedis**.rpush(**"SSM"**, **"Mybatis"**);  
**jedis**.lrange(**"SSM"**, 0, -3).forEach(**logger**::info);

使用Java操作Redis的集合：

**jedis**.del(**"user"**);  
**jedis**.sadd(**"user"**, **"刘景行"**);  
**jedis**.sadd(**"user"**, **"刘楚兴"**);  
**jedis**.sadd(**"user"**, **"梅新养"**);  
**jedis**.smembers(**"user"**).forEach(**logger**::info);  
**logger**.info(**"是否包含："**+**jedis**.sismember(**"user"**, **"梅新养"**));  
**jedis**.srem(**"user"**, **"刘景行"**); *//移除元素***logger**.info(**"set长度："**+**jedis**.scard(**"user"**));

使用Java操作Redis事务：

**jedis**.watch(**"num"**);  
Transaction transaction = **jedis**.multi();  
**for** (**int** i=0; i < 5; i++){  
 transaction.set(**"num"**, **""**+i); *//事务中不能使用jedis对象*}  
transaction.exec().forEach(item -> **logger**.info(item.toString()));

在Java中使用Redis连接池：

**public final class** RedisUtil {  
  
 *//Redis服务器IP* **private static** String *ADDR* = **"192.168.0.100"**;  
  
 *//Redis的端口号* **private static int** *PORT* = 6379;  
  
 *//访问密码* **private static** String *AUTH* = **"admin"**;  
  
 *//可用连接实例的最大数目，默认值为8；  
 //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(耗尽)。* **private static int** *MAX\_ACTIVE* = 1024;  
  
 *//控制一个pool最多有多少个状态为idle(空闲的)的jedis实例，默认值也是8。* **private static int** *MAX\_IDLE* = 200;  
  
 *//等待可用连接的最大时间，单位毫秒，默认值为-1，表示永不超时。如果超过等待时间，则直接抛出JedisConnectionException；* **private static int** *MAX\_WAIT* = 10000;  
  
 **private static int** *TIMEOUT* = 10000;  
  
 *//在borrow一个jedis实例时，是否提前进行validate操作；如果为true，则得到的jedis实例均是可用的；* **private static boolean** *TEST\_ON\_BORROW* = **true**;  
  
 **private static** JedisPool *jedisPool* = **null**;  
  
 */\*\*  
 \* 初始化Redis连接池  
 \*/* **static** {  
 **try** {  
 JedisPoolConfig config = **new** JedisPoolConfig();  
 config.setMaxActive(*MAX\_ACTIVE*);  
 config.setMaxIdle(*MAX\_IDLE*);  
 config.setMaxWait(*MAX\_WAIT*);  
 config.setTestOnBorrow(*TEST\_ON\_BORROW*);  
 *jedisPool* = **new** JedisPool(config, *ADDR*, *PORT*, *TIMEOUT*, *AUTH*);  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
 }  
  
 */\*\*  
 \* 获取Jedis实例  
 \** ***@return*** *\*/* **public synchronized static** Jedis getJedis() {  
 **try** {  
 **if** (*jedisPool* != **null**) {  
 Jedis resource = *jedisPool*.getResource();  
 **return** resource;  
 } **else** {  
 **return null**;  
 }  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 **return null**;  
 }  
 }  
  
 */\*\*  
 \* 释放jedis资源  
 \** ***@param jedis*** *\*/* **public static void** returnResource(**final** Jedis jedis) {  
 **if** (jedis != **null**) {  
 *jedisPool*.returnResource(jedis);  
 }  
 }  
}