**配置环境变量：**

vim ~/.bash\_profile

然后添加环境变量

export REDIS\_HOME=/usr/local/redis

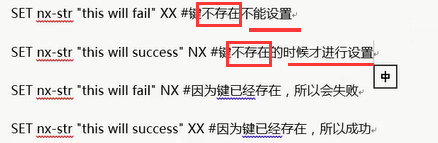
export PATH=$PATH:$REDIS\_HOME/bin

source ~/.bash\_profile使其生效；

**配置完之后，就可以在任何地方使用redis-server和redis-cli**

Redis中默认有16个数据库，索引为0-15，默认是第0个

通过select 索引值 可以切换数据库



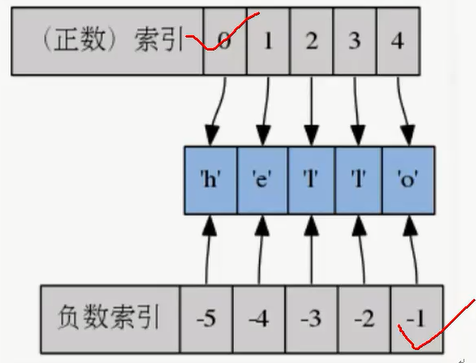
**Mset/mget同时设定或获取多个值**

set myphone “myphone”

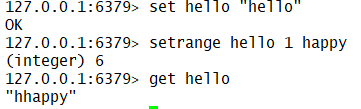
append myphone “----”

**就是将两个字符串进行拼接**

**索引**



**SETRANGE命令**我们可以从索引index开始，用你想写入的value值替换掉给定的键key所存储的字符串中的部分，只接受正数索引





**左闭又闭**

**数字操作**



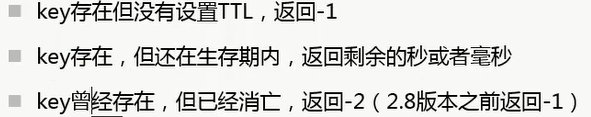
INCRBY KEY NUM

DECRBY KEY NUM

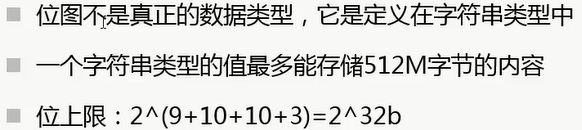
INCRBYFLOAT KEY NUM，浮点数没有负数，可采用INCRBYFLOAT KEY 负数实线减法

**当用户针对一个数字进行字符串的命令操作的时候，会先将数字值转换为字符串，然后在执行命令**。

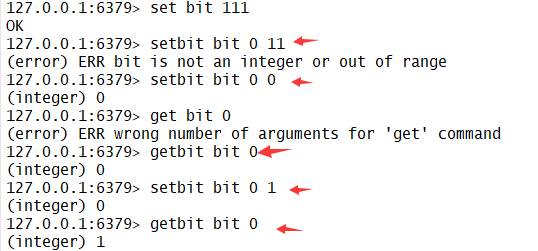
TTL key：

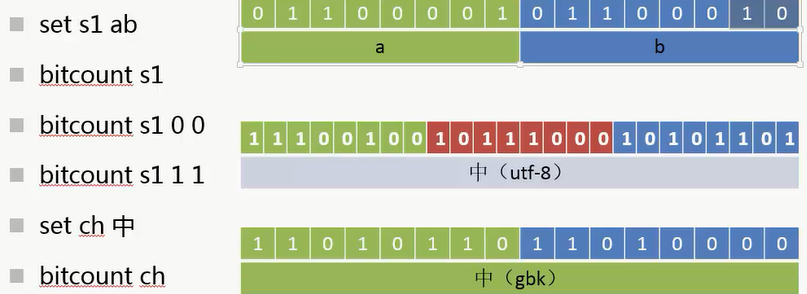


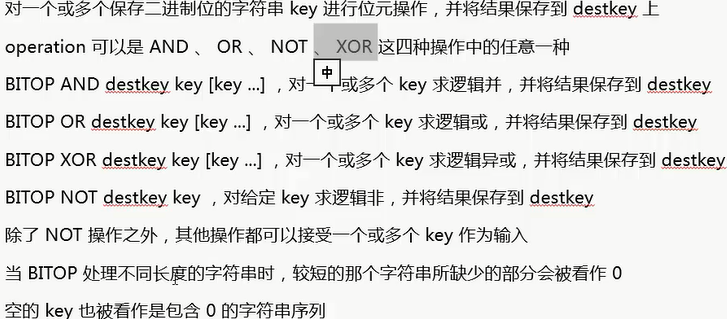
**Bitmap位图**



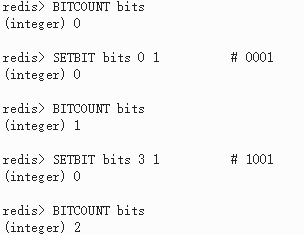




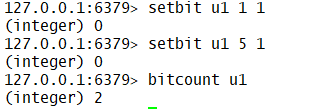




Bitcount统计指定位区间上**值为1**的个数

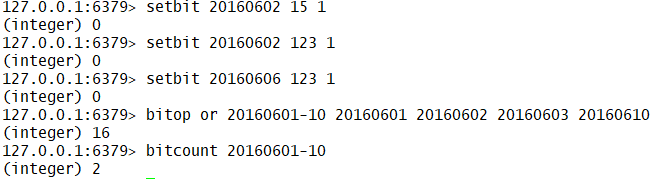


**统计用户上线的次数，u1 1 1表示u1用户第一天上线，u1 5 1表示u1用户第五天上线**



**按天统计网站活跃用户，求6月1日到6月10日**

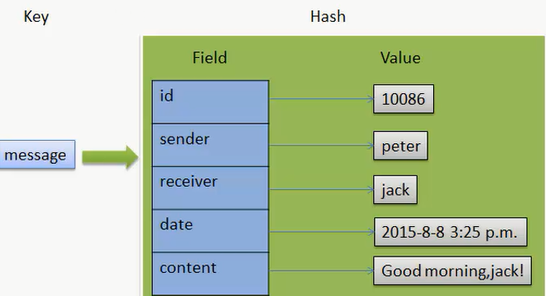
天作为key，用户id作为offset，上线置为1；



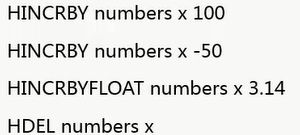
Hash散列

由field和关联的value组成的map键值对

一个hash中最多包含2^32-1键值对



在字段对应的值上进行整数的增量计算

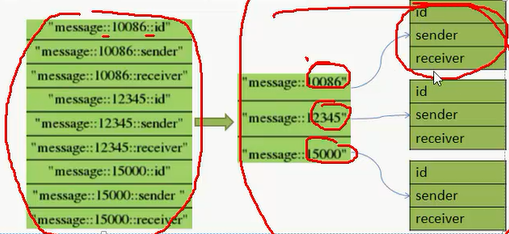


**Hash用途**

节约内存空间

每创建一个键，它都会为这个键存储一些附加管理信息（比如这个键的类型，这个键最后一次被访问的时间等等）

所以数据库里面的键越多，redis数据库服务器在存储附加管理信息方面耗费的内存就越多，，在字段对应的值上进行浮点数的增量计算就需更多的cpu支持



**不适合hash的情况**

使用二进制位操作命令：因为redis目前只支持对字符串进行setbit，getbit，bitop等操作，如果你想使用这些操作，那么只能使用字符串key，

使用过期键功能：redis的键过期功能目前只能对键进行过期操作，而不能对散列的字段进行过期操作，因此如果要对键值对数据使用过期功能的话，那么只能把**键值对**存储在**字符串**中。

**微博好友的关注**

用户id为key，field为好友id，value为关注时间

Key field value

User:1000 user:606 20150808

用户的维度统计

统计数：关注数follow，粉丝数，喜欢的商品数，发帖数

用户为key，不同维度为field，value为统计数

比如关注了5人

HSET user:100000 follow 5

**新浪微博的共同关注：**

需求：当用户访问另一个用户的时候，会显示两个用户共同关注哪些相同的用户

设计：将每个用户关注的用户放在集合中，求交集即可

实现如下：

Peter={‘john’, ‘jack’, ‘may’}

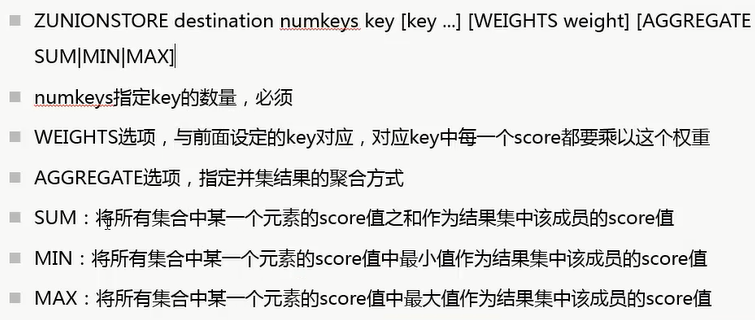
Ben={‘john’, ‘jack’, ‘tom’}

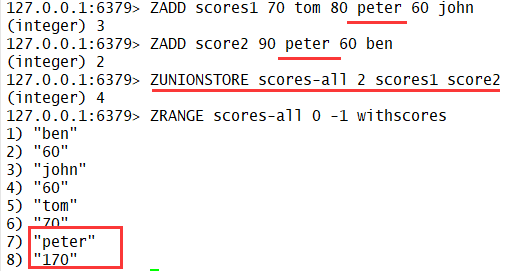
那么peter和ben的共同关注：

SINTER peter ben 结果为{‘john’, ‘jack’}

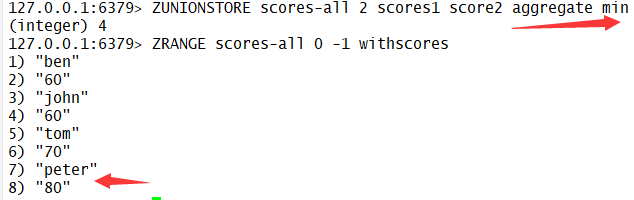
Sorted set

并集

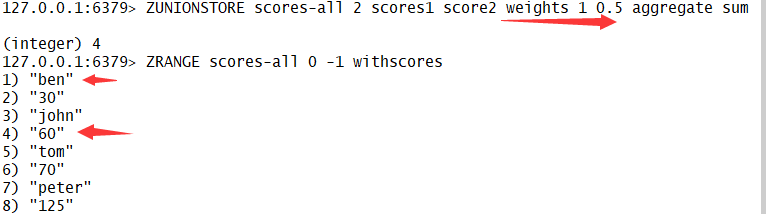




说明如果后面不加参数，说明默认使用sum模式



聚合使用的是min，即取相同key中的score的较小的那个



这个表示，求并集的时候，scores1中的score\*1 ，scores2中的score\*0.5，其他不变

交集与并集使用方式基本一致

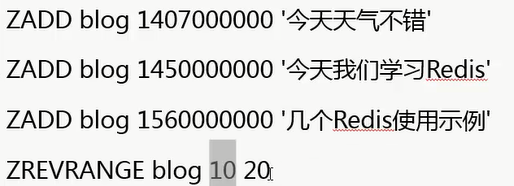
**使用：**

**网易云音乐排行榜**

**新浪微博翻页**

新闻网站、博客、论坛、搜索引擎，页面列表较多，需要分页

Blog这个key中使用时间戳作为score



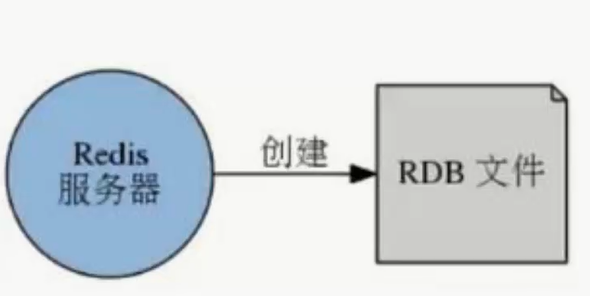
博客blog、论坛bbs这些作为key，然后时间戳作为score；

Zrevrange blog 10 20就是去前10到20条的数据

**Redis的持久化**

**RDB持久化**

将服务器包含的所有数据库数据以**二进制文件**的形式保存到硬盘里面



在redis服务器创建RDB文件的情况中，以下三种情况最常见：

**手动**

**1.服务器执行客户端发送的save命令；**

**2.服务器执行客户端发送的bgsave命令；**

**自动**

**3.使用save配置选项设置的自动保存条件被满足，服务器自动执行bgsave**

这三种创建RDB文件的情况中，前两种需要用户手动执行，而第三种情况则是**由redis服务器**自动执行。

**手动创建rdb文件，也就是手动发送save命令或者bgsave命令**

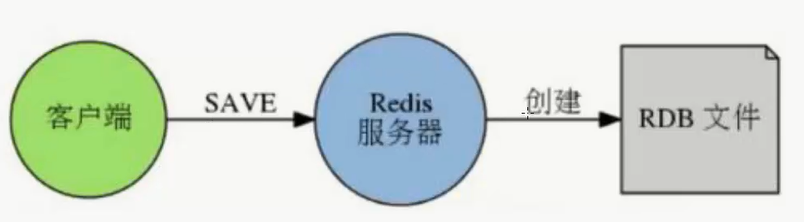
通过使用客户端向服务器发送save命令，可以命令服务器去创建一个新的rdb文件：

Redis>save

在执行save命令的过程中（即时创建rdb文件的过程中），redis服务器将**被阻塞**，无法处理客户端发送的命令请求，只有在save命令执行完毕之后（即rdb文件创建完毕之后），服务器才会重新开始处理客户端发送的命令请求。

如果rdb文件已经存在，那么服务器将自动使用心得RDB文件去代替旧的rdb文件

**平常可以使用时间来命名rdb文件，这样以后需要使用就加载某个时间的rdb文件**



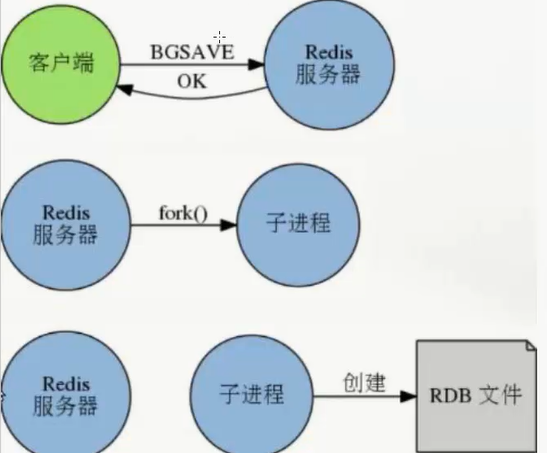
--执行bgsave命令同样可以创建一个新的rdb文件，这个命令和save命令的区别在于，bgsave命令不会造成redis服务器阻塞：在执行bgsave过程中，redis服务器仍然可以正常的处理其他客户端发送的命令请求

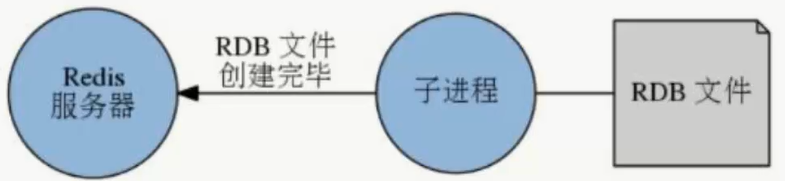
--bgsave命令不会造成服务器阻塞的原因在于：

----当redis服务器接收到bgsave命令的时候，它不会自己创建rdb文件，而是通过fork来生成一个子进程，然后由子进程负责创建rdb文件，而自己则继续处理客户端的命令请求；

----当子进程创建好的rdb文件并退出时，它会向父进程（即负责处理命令请求的redis服务器）发送一个信号，告知它rdb文件已经创建完毕；

----最后redis服务器（父进程）结束子进程创建的rdb文件bgsave执行完毕





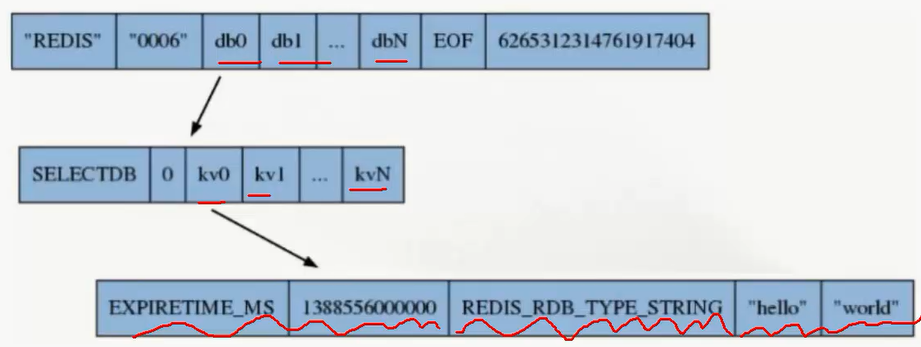
创建子进程，会消耗额外的内存，所以save创建rdb文件的速度会比bgsave快，可以集中资源来创建rdb文件，save和bgsave没有孰好孰坏之说，如果数据库正在上线中，那当然就是使用bgsave，如果要维护，在凌晨3点，那最好使用save，比如维护需要停机一个消失，或者半个小时，那你使用save命令，这是阻塞了也没事，使用save命令好点。

**自动去创建rdb文件**

Save 300 10

表示“如果距离上一次创建rdb文件已经过去300秒，并且服务器的所有数据库总共已经发生了不少于10次修改，那么执行bgsave命令”

Rdb文件

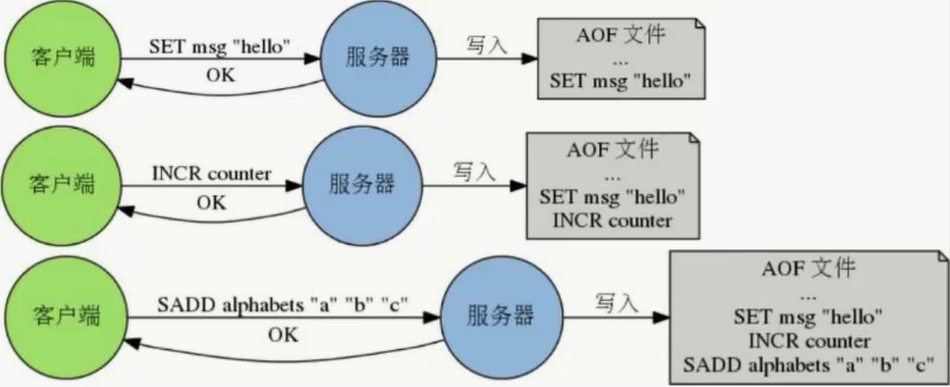


数据库-键值对集合-每个键值对的信息

**AOF持久化**

每当有修改的数据库的命令被执行时，服务器就会将执行的命令写入到aof文件的末尾。

因为aof文件里面存储了服务器执行过的所有数据库修改的命令，所以给定一个aof文件，服务器只要重新执行一遍aof里面的命令，就可以达到还原数据库的目的了



目前常见的操作系统中，执行系统调用write函数，将一些内容写入到某个文件里面，为了提高效率，系统通常不会直接将内容写入到硬盘里面，而是先将内容放入到一个内存缓冲区（buffer）里面，等到缓冲区被填满，或者用户执行fsync调用和fdatasync调用时才将存储在缓冲区中的内容真正写入到硬盘里。

为了控制redis服务器在遇到意外停机而丢失掉数据，redis为aof持久化提供了appendfsync选项，这个选项的值可以是always、everysec或者no，这些值的意思分别为：

Always：服务器每写入一个命令，就调用一次fdatasync，将缓冲区里面的命令写入到硬盘里面，在这种模式下，服务器即使遭遇意外停机，也不会丢失任何已经成功执行的命令数据。

Everysec：服务器每一秒调用一次fdatasync，将缓冲区里面的命令写入到键盘李米娜，在这种模式下，服务器即使遭遇意外停机时，最多只丢失一秒钟内的执行的命令数据。

No：服务不主动调用fdatasync，由操作系统决定任何将缓冲区的命令写入到硬盘里面，在这种模式下，服务器遭遇意外停机时，丢失命令的数量是不确定的

默认值：everysec

为了让aof文件的大小控制在合理范围，避免它胡乱增长，redis提供了**aof重写功能**，通过这个功能，服务器可以产生一个新的aof文件：

--新的aof文件记录的数据库数据和原有aof文件记录的数据库数据完全一样

--新的aof文件会使用尽可能少的命令来记录数据库的数据，因此新的aof文件的体积通常会比原有aof的体积小很多

--aof重写期间，服务器不会被阻塞，可以正常处理客户端发送的命令请求

**重写过程：**

在重写即将开始之际，**redis会创建（fork）一个“重写子进程”**，这个子进程会首先读取**现有的AOF文件**，并将其包含的指令进行**分析压缩并写入到一个临时文件**中。

与此同时，主工作进程会将新接收到的写指令一边**累积到内存缓冲区**中，一边**继续写入到原有的AOF文件**中，这样做是保证**原有的AOF文件的可用性**，避免在**重写过程中出现意外**。

当“重写子进程”完成**重写工作**后，它会给**父进程发一个信号**，父进程收到信号后就会将**内存中缓存的写指令追加到新AOF文件中**。

当追加结束后，redis就会用新AOF文件来代替旧AOF文件，之后再有新的写指令，就都会追加到新的AOF文件中了。

**子进程读取旧的aof文件，然后分析压缩到新的aof；**

**同时主进程对新的写指令操作，既放到内存缓冲区，又放到旧的aof文件中，防止重写失败，保证旧aof文件的可用性；**

**等到子进程重写结束了，给父进程信号，会将内存缓冲区的指令追加到新的aof文件中，然后覆盖旧的aof文件。**

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

重写示例



两种方法能出发aof重写

1.客户端向服务器发送BGREWRITEAOF命令。

2.通过设置配置选项来让服务器自动执行BGREWRITEAOF命令，他们分别是：

Auto-aof-rewrite-min-size<size>, 触发aof重写所需的最小体积：只要在aof文件的体积大于等于size时；

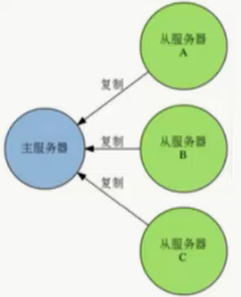
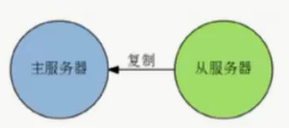
**Redis主从复制**

主从复制replication

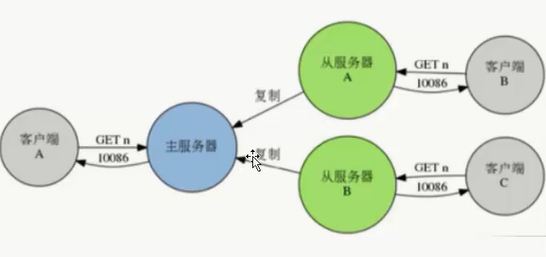
--redis服务可以有多个该服务的复制品，这个redis服务成为master，其他复制品成为slaves

--只要网络连接正常，master会一直将自己的数据更新同步给slaves，保持主从同步

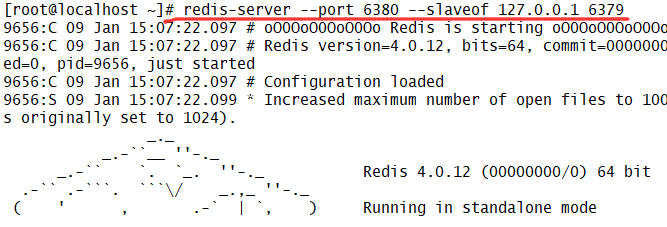
--只有master可以执行写命令，slaves只能执行读命令



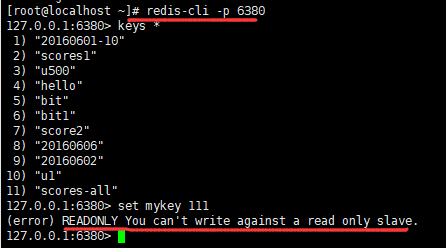
--客户端可以连接slaves执行请求，来降低master的读压力



**主从复制的创建**

第一种方法

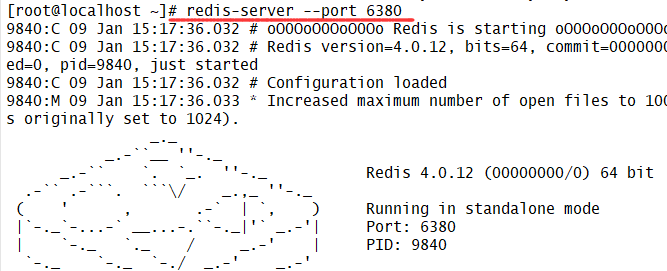
使用redis-cli –p 6380 连接指定端口号的redis

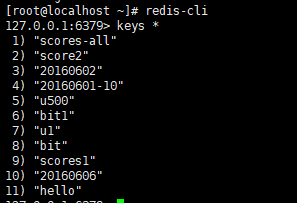
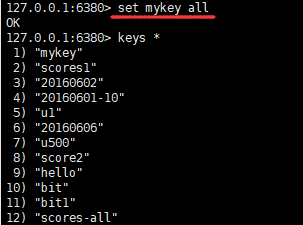


第二种方法：直接使用redis-server –port 6380，以master启动6380的redis

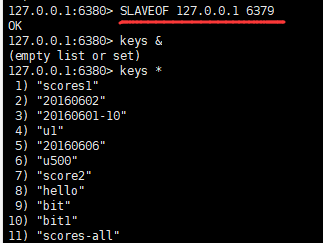
然后使用slaveof 127.0.0.1 6379，将服务器转换为slave

使用slaveof no one， 将服务器重新恢复到master，不会丢弃已同步的数据





将其设为从服务器之后就会去同步主服务器，主服务器是6379，6379中没有mykey，所以从服务器也没有mykey。



**主从复制问题**

--一个master可以有多个slaves

--slave下线，只是读请求的处理性能下降

--master下线，写请求无法执行

--其中一台slave使用slaveof no one 命令成为master，其他slaves执行slaveof命令指向这个新的master，从它这里同步数据

以上过程都是手动的，能够实现自动，这就需要sentinel哨兵，实线故障转移failover操作

**Redis哨兵**

**高可用sentinel**

--官方提供的高可用方案，可以用它管理多个redis服务实例

--编译后产生redis-sentinel程序文件

--redis sentinel是一个分布式系统，可以在一个架构中运行多个sentinel进程

**启动sentinel**

--将src目录下产生的redis-sentinel文件复制到bin目录下

--启动一个运行在sentinel模式下的redis服务实例

----redis-sentinel

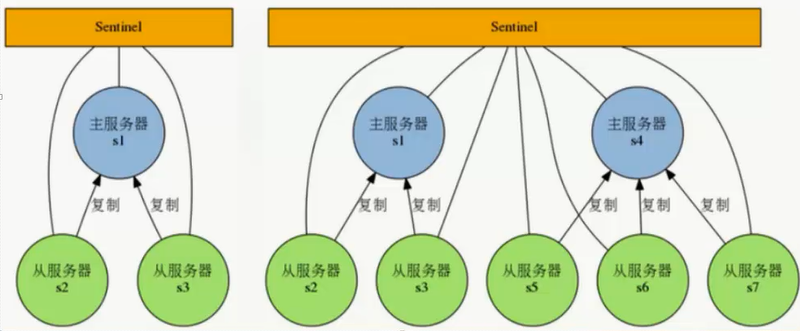
----redis-server /path/to/sentinel.conf –sentinel

--redis sentinel是一个分布式系统，可以在一个架构中运行多个sentinel进程

**监控monitoring**

--sentinel会不断检查master和slaves是否正常

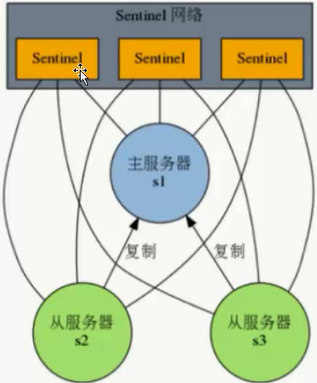
--每个sentinel 可以监控任意多个master和该master下的slaves



**Sentinel网络**

--监控同一个master的sentinel会自动连接，组成一个分布式的sentinel网络，互相通信并交换彼此关于被监视服务器的信息

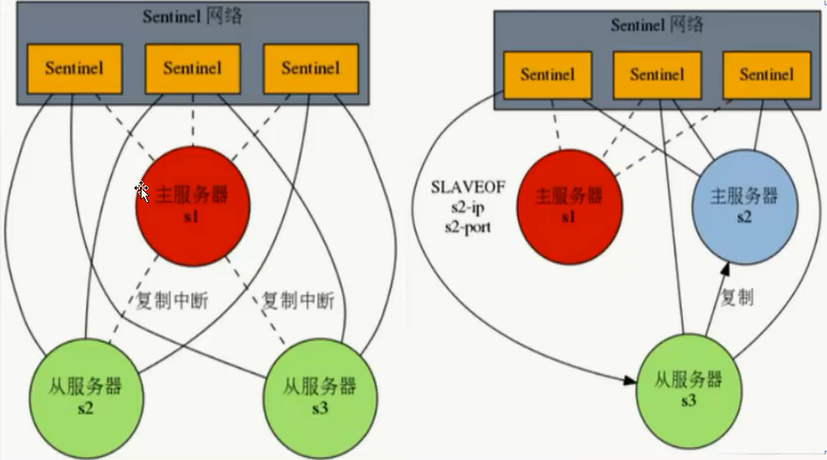
下图中，3个sentinel监控着s1和它的2个slave



**服务器下线**

--当一个sentinel认为被监视的服务器已经下线的时候，它会向网络中的其他sentinel进行确认，判断该服务器是否真的下线

--如果下线的服务器为主服务器，那么sentinel网络将对下线的主服务器进行自动故障转移，通过将下线的主服务器的某个从服务器提升为新的主服务器，其他从服务器之后就复制新的主服务器。



**Sentinel配置文件**

--至少包含一个监控配置选项，用于指定被监控master的相关信息

--sentinel monitor mymaster 127.0.0.1 6379 2，监视mymaster的主服务器，服务器ip和端口，将这个主服务器判断为下线失效了，至少需要**2个sentinel**同意，如果**多数sentinel同意**，才会执行故障转移

--sentinel会根据master的配置自动发现master的slaves

--sentinel默认端口26379

**Sentinel总结**

--主从复制，解决了度请求的分担，从节点下线，会使得读请求能力有所下降

--master只有一个，写请求单点问题

--sentinel会在master下线后自动执行failover操作，提升一台slave为master，并让其它slaves重新成为新的master的slaves

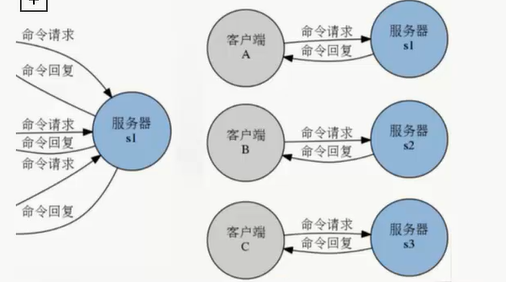
--主从复制+哨兵sentinel只解决了读性能和高可用问题，但是**没有解决写性能问题**

**问题引出----twemproxy**

--主从对写压力没有分担

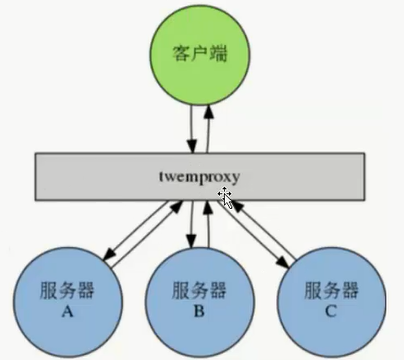
--解决思路：使用多个节点分担，将写请求分散到不同节点处理

--分片sharding：多节点分担的思路就是关系性数据库处理大表的水平切分思路



**Twemproxy**

--twitter开发，代理用户的读写请求



--twitter开发的代理服务器，允许用户将多个redis服务器添加到一个服务器pool里面，并通过用户选择的**散列函数和分布函数**，将来自客户端的命令请求分发给服务器池中的各个服务器。

--通过twemproxy我们可以将数据库分片到多台redis服务器上面，并使用这些服务器来分担系统压力以及数据库容量：在服务器硬件条件相同的情况下，对于一个包含n台redis服务器的池来说，池中每台平均1/N的客户端命令请求

--向池里添加更多服务器可以线性的扩展系统处理命令请求的能力，以及系统能够保存的数据量。

**总结：**

--前端使用twemproxy做代理，后端的redis数据能基本上根据key来进行比较均衡的分布

--后端一台redis挂掉后，twemproxy能够自动摘除，恢复后，twemproxy能够自动识别、恢复并重新加入到redis组中重新使用

--redis挂掉后，后端数据是否丢失依据redis本身的持久化策略配置，与twemproxy基本无关

--如果要新增加一台redis，twemproxy需要重启后才能生效；并且数据不会自动更新，需要人工单独写脚本来实现

--原来已经有2个节点redis，后续增加2个redis，则数据分布计算与原来的redis分布无关，现有数据如果需要分布均匀的话，需要人工单独处理

--如果twemproxy的后端节点数量发生变化，twemproxy相同算法的前提下，原来的数据必须重新处理分布，否则会存在找不到key值的情况

--不管twemproxy后端有几台redis，前端的单个twemproxy的性能最大也只能和单台redis性能差不多

--如同时部署多台twemproxy配置一样，客户端分别连接多台twemproxy可以在一定条件下提高性能

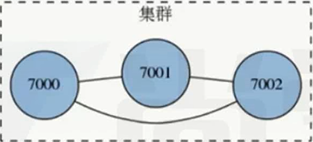
**Redis集群**

--3.0以上支持

--由多个redis服务器组成的分布式网络服务集群

每个redis服务器成为节点node，节点之间会互相通信，两两相连

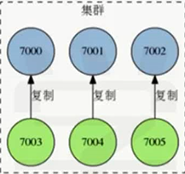
--redis集群无中心节点



**Redis集群节点复制**

--Redis集群的每个节点都有两种角色可选：主节点master node、从节点slave node。其中主节点用于存储数据，而从节点则是某个主节点的复制品

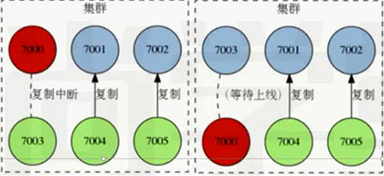
--当用户需要处理更多的读请求的时候，添加从节点可以扩展系统的读性能，因为redis集群重用了单机redis复制特性的代码，所以集群的复制行为和我们之前介绍的单机复制特性的行为是完全一样的。



**Redis集群故障转移**

--redis集群的主节点内置了类似redis sentinel的节点故障检测和自动故障转移功能，当集群中的某个主节点下线时，集群中的其他在线主节点会注意到这一点，并对已下线的主节点进行故障转移

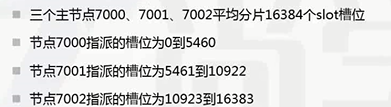
--集群进行故障转移的方法和redis sentinel进行故障转移的方法基本一样，不同的是，在集群方面，故障转移是由集群中其他在线的主节点负责进行的，所以集群不必另外使用redis sentinel



**Redis集群 的分片**

--集群将整个数据库分为16384个槽位slot，所有key都属于这些slot中的一个，key的槽为计算公式为slot\_number=crc16(key) % 16384，其中crc16为16位的循环冗余校验和函数

--集群中的每个主节点都可以处理0-16384个槽，当16384个槽都有某个节点在负责处理时，集群进入**上线状态**，并开始处理客户端发送的数据命令请求。

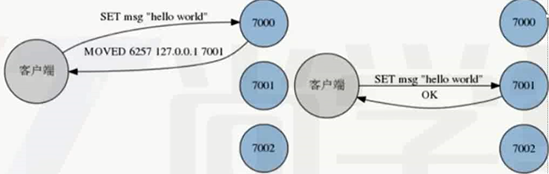


**Redis集群redirect转向**

--由于redis集群无中心节点，请求会发给任意主节点

--主节点只会处理自己负责槽位的命令请求，其他槽位的命令请求，该主节点会返回客户端一个转向错误

--客户端根据错误中包含的地址和端口重新向正确负责的主节点发起命令请求



**Redis集群搭建**

--创建多个主节点

--为每一个节点指派slot，将多个节点连接起来，组成一个集群

--槽位分片完成后，集群进入上线状态

--6个节点：3个主节点，每一个主节点有一个从节点