

高并发核心技术Redis

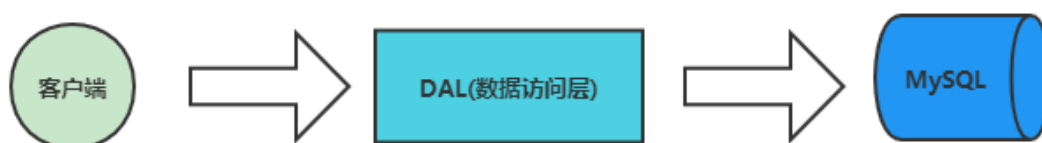
一、Redis概述

1 NoSQL介绍

1.1 NoSQL由来

任何技术的出现都是一步一步演进出来的。

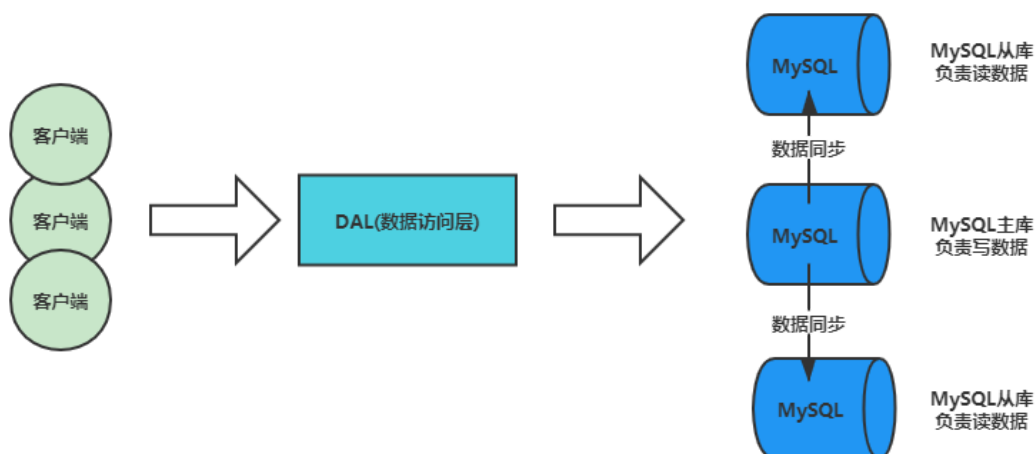
1. 在互联网诞生初期，一个网站一个应用访问量都不大，使用单机MySQL数据库可以应对。



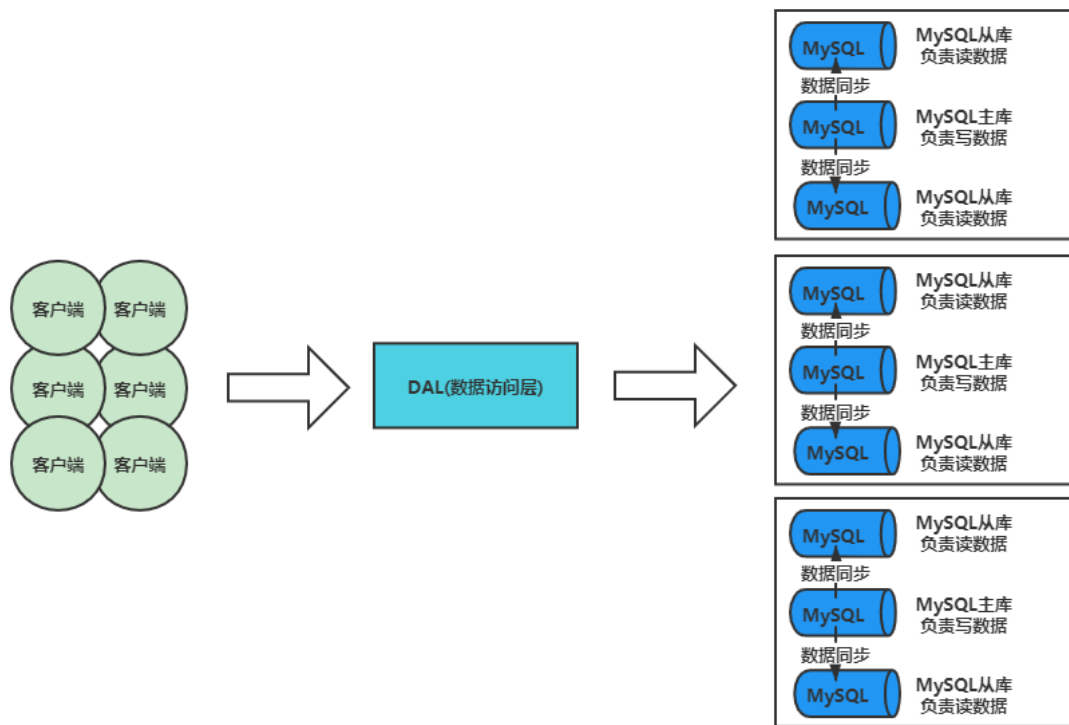
2. 随着用户的不断增多，单机MySQL可能无法放下，服务器也可能承受不住

一个网站，大多数的情况都是查询，为了减少数据库的压力，引入读写分离策略，让主数据库处理事务性增、改、删操作，而从数据库处理SELECT查询操作。

通过读写分离的方式，解决问题。

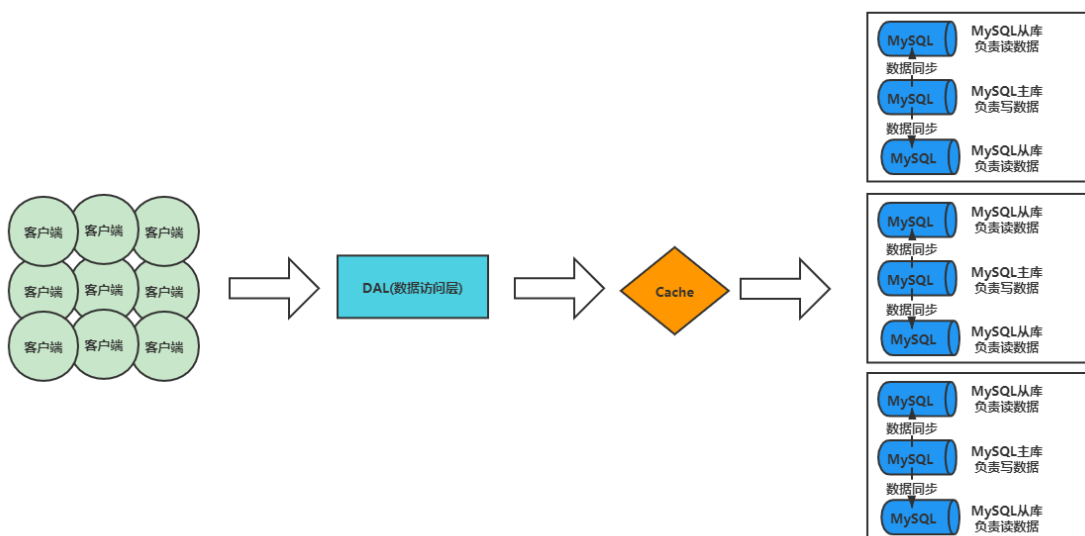


3. Mysql读的问题得到了缓解，写的压力依然存在，开始通过分库分表和MySQL集群来解决写的问题。



4. 当前，数据量越来越大、数据类型也越来越复杂，Mysql数据库在面对数据量很多、变化很大数据，如大文本字段、图片等，MySQL效率就低了，并且扩展性差，大数据量场景下IO压力大，结构更改困难。如果有一种数据库专门来存储此类数据，分担读取数据的压力，MySQL压力就会变得很小。

NoSQL数据库就应运而生，可以很好的解决上述问题。



1.2 什么是NoSQL

NoSQL = Not Only SQL 不仅仅是SQL

泛指非关系型数据库，对应对比关系型数据库，NoSQL没有sql语句使用，不需要设计表结构，没有表关系。

很多数据类型如：用户的个人信息，社交网络，地理位置，这些数据类型的存储不需要一个固定的模式，类似于Map<String,Object>使用键值对来控制。

1.3 NoSQL的特点

- 1. 方便扩展（数据之间没有关系，很好扩展）
- 2. 大数据量高性能（Redis一秒写8万次，读取11万次，NoSQL缓存记录级，是一种细粒度的缓存，性能比较高）
- 3. 数据类型多样性（不需要事先设计数据库，如果是数据量特别大的表，设计繁琐）
- 4. 传统RDBMS 与 NoSQL

RDBMS:

- 结构化数据
- 结构化查询语句SQL
- 数据和关系都存储在单独的表中
- 数据操纵语言,数据定义语言
- 严格的一致性
- 基础事务

NoSQL:

- 没有固定的查询语言
- 键值对存储、列存储、文档存储、图关系数据库
- 最终一致性
- CAP定理和BASE理论
- 高性能，高可用，高可扩展

1.4 NoSQL的四大分类

分类	常见数据库	典型应用场景	优点	缺点
键值对 (key-value)	Tokyo Cabinet/Tyrant, Redis , Voldemort, Oracle BDB	内容缓存，主要用于处理大量数据的高访问负载，也用于一些日志系统等。	查找速度快	数据无结构化，通常只被当作字符串或者二进制数据
文档型数据库	CouchDB, MongoDB	Web应用（与Key-Value类似，Value是结构化的，不同的是数据库能够了解Value的内容）	数据结构要求不严格，表结构可变，不需要像关系型数据库一样需要预先定义表结构	功能相对局限
列存储数据库	Cassandra, HBase , Riak	分布式的文件系统	查找速度快，可扩展性强，更容易进行分布式扩展	查询性能不高，而且缺乏统一的查询语法。
图形数据库	Neo4J, InfoGrid, Infinite Graph	社交网络，推荐系统等。专注于构建关系图谱	利用图结构相关算法。比如最短路径寻址，N度关系查找等	很多时候需要对整个图做计算才能得出需要的信息，而且这种结构不太好做分布式的集群方案。

2 Redis概述

2.1 Redis是什么

1. Redis (Remote Dictionary Server 远程字典服务) 是一个开源的使用ANSI C语言编写、支持网络、内存亦可持久化的key-value数据库，并提供多种语言的API。
2. Redis是一个**key-value**存储系统，它支持存储的value类型相对更多，包括**string**、**list**、**set**、**zset** (sorted set --有序集合) 和**hash**。这些数据结构都支持push/pop、add/remove及取交集并集和差集及更丰富的操作，而且这些操作都是原子性的。在此基础上，Redis支持各种不同方式的排序。为了保证效率，数据都是缓存在内存中，Redis会周期性的把更新的数据写入磁盘或者把修改操作写入追加的记录文件，并且在此基础上实现了master-slave (主从) 同步。
3. Redis提供了java、C/C++、PHP、JavaScript、Perl、Object-C、Python、Ruby、Erlang等客户端，使用很方便。

2.2 Redis能干嘛

1. 读写效率高，用于高速缓存
2. 发布，订阅消息 (消息通知)
3. 地图信息分析
4. 活动排行榜或计数
5.

2.3 Redis特点

1. 多样的数据类型：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。
2. Redis支持数据的持久化，可以将内存中的数据保存在磁盘中，重启的时候可以再次加载进行使用。
3. Redis的所有操作都是原子性的。
4. 支持主从复制及集群。

3 Redis安装

3.1 下载地址

Redis官方网址：<https://redis.io/>

3.2 下载Redis

1. 下载6.2.4 for Linux

Redis is an open source (BSD licensed), in-memory data structure store, used as a database, cache, and message broker. Redis provides data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperloglogs, geospatial indexes, and streams. Redis has built-in replication, Lua scripting, LRU eviction, transactions, and different levels of on-disk persistence, and provides high availability via Redis Sentinel and automatic partitioning with Redis Cluster. [Learn more →](#)

Try it

Ready for a test drive? Check this [interactive tutorial](#) that will walk you through the most important features of Redis.

Download it

Redis 6.2.4 is the latest stable version.

Interested in release candidates or unstable versions? [Check the downloads page](#).

Quick links

Follow day-to-day Redis on [Twitter](#) and [GitHub](#). Get help or help others by subscribing to [our mailing list](#), we are 5,000 and counting!

2. redis-6.2.4.tar.gz上传至CentOS并解压，解压后得到redis-6.2.4目录

解压命令：

```
tar -zxvf redis-6.2.4.tar.gz
```

3.3 安装GCC

安装C语言编译环境

```
dnf group install "Development Tools"
```

通过使用 `gcc --version` 命令打印 GCC 版本，来验证 GCC 编译器是否被成功安装：

```
gcc --version
```

3.4 安装Redis

1. 编译Redis

在redis-6.2.4目录下执行：

```
make
```

出现下图代表编译成功

```

CC gopher.o
CC tracking.o
CC connection.o
CC tls.o
CC sha256.o
CC timeout.o
CC setcpuaffinity.o
CC monotonic.o
CC mt19937-64.o
LINK redis-server
INSTALL redis-sentinel
CC redis-cli.o
CC cli_common.o
LINK redis-cli
CC redis-benchmark.o
LINK redis-benchmark
INSTALL redis-check-rdb
INSTALL redis-check-aof

Hint: It's a good idea to run 'make test' ;)

make[1]: 离开目录"/download/redis-6.2.4/src"

```

2. 安装Redis

在redis-6.2.4目录下执行：

```
make install
```

出现下图代表安装成功

```

[root@localhost redis-6.2.4]# make install
cd src && make install
make[1]: 进入目录"/download/redis-6.2.4/src"
CC Makefile.dep

Hint: It's a good idea to run 'make test' ;)

INSTALL redis-server
INSTALL redis-benchmark
INSTALL redis-cli
make[1]: 离开目录"/download/redis-6.2.4/src"

```

3. 安装目录： /usr/local/bin

```

[root@localhost bin]# cd /usr/local/bin/
[root@localhost bin]# ls
redis-benchmark  redis-check-aof  redis-check-rdb  redis-cli  redis-sentinel  redis-server
[root@localhost bin]#

```

redis-benchmark: Redis自带的基准性能测试工具

redis-check-aof: 对有问题 AOF 文件进行修复，AOF和RDB文件后面会说明

redis-check-rdb: 对有问题 RDB文件进行修复

redis-sentinel: Redis集群使用

redis-cli: 客户端

redis-server: 服务器启动

4. 服务启动

前台启动： /usr/local/bin下执行

```
./redis-server
```

```

[root@localhost bin]# ./redis-server
64830:C 28 Jun 2021 06:20:16.361 # 000000000000 Redis is starting 000000000000
64830:C 28 Jun 2021 06:20:16.361 # Redis version=6.2.4, bits=64, commit=00000000, modified=0, pid=64830, just started
64830:C 28 Jun 2021 06:20:16.361 # Warning: no config file specified, using the default config. In order to specify a config file use ./redis-server /path/to/redis.conf
64830:M 28 Jun 2021 06:20:16.361 * Increased maximum number of open files to 10032 (it was originally set to 1024).
64830:M 28 Jun 2021 06:20:16.361 * monotonic clock: POSIX clock_gettime

Redis 6.2.4 (00000000/0) 64 bit

Running in standalone mode
Port: 6379
PID: 64830

https://redis.io

64830:M 28 Jun 2021 06:20:16.362 # WARNING: The TCP backlog setting of 511 cannot be enforced because /proc/sys/net/core/somaxconn is set to the lower value of 128.
64830:M 28 Jun 2021 06:20:16.362 # Server initialized
64830:M 28 Jun 2021 06:20:16.362 # WARNING overcommit_memory is set to 0! Background save may fail under low memory condition. To fix this issue add 'vm.overcommit_memory = 1' to /etc/sysctl.conf and then reboot or run the command 'sysctl vm.overcommit_memory=1' for this to take effect.
64830:M 28 Jun 2021 06:20:16.362 * Loading RDB produced by version 6.2.4
64830:M 28 Jun 2021 06:20:16.362 * RDB age 6 seconds
64830:M 28 Jun 2021 06:20:16.362 * RDB memory usage when created 0.77 Mb
64830:M 28 Jun 2021 06:20:16.362 * DB loaded from disk: 0.000 seconds
64830:M 28 Jun 2021 06:20:16.362 * Ready to accept connections

```

后台启动:

拷贝redis-6.2.4目录中的redis.conf文件到其他目录

```

mkdir /usr/local/myredis
cp redis.conf /usr/local/myredis/redis.conf

```

设置/usr/local/myredis/redis.conf文件中的**daemonize**属性, 由no改为yes

在/usr/local/bin下执行

```
./redis-server /usr/local/myredis/redis.conf
```

通过ps aux | grep redis-server查看服务是否启动

```

[root@localhost bin]# ps aux | grep redis-server
root      65187  0.0  0.4 62184 3240 ?        Ssl  06:49   0:00 ./redis-server 127.0.0.1:6379
root      65241  0.0  0.1 12348 1120 pts/1    R+   06:51   0:00 grep --color=auto redis-server
[root@localhost bin]#

```

5. 客户端启动

/usr/local/bin下执行

```
./redis-cli
```

```

[root@localhost bin]# ./redis-cli
127.0.0.1:6379>

```

ping命令可以检测服务器是否正常 (服务器返回PONG)

```
ping
```

```
[root@localhost bin]# ./redis-cli
127.0.0.1:6379> ping
PONG
```

4 Redis基本知识

1. 端口6379的由来

6379 = Merz



Merz全名Alessia Merz，是意大利的一位广告女郎。



2. 默认有16个数据库，且初始状态默认选择0号数据库(即第一个数据库)。
3. 可以使用select 进行数据库切换。

select 8 切换到8号数据库

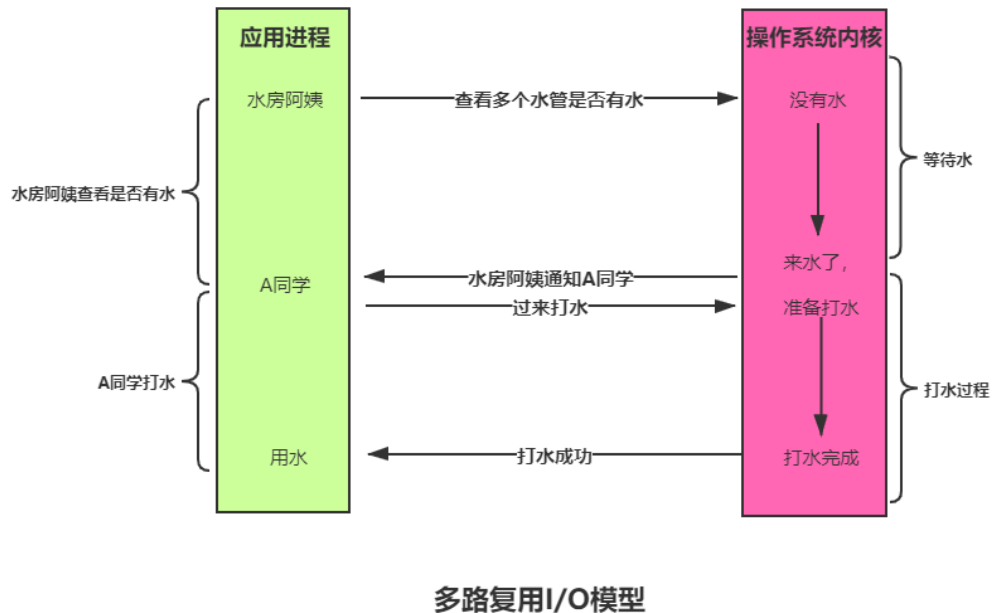
```
127.0.0.1:6379> select 8
OK
127.0.0.1:6379[8]>
```

4. 统一密码管理，所有库密码都一致。
5. dbsize-查看当前数据库的key数量。
6. flushdb-清空当前库。

7. flushall-清空所有库。

8. 为什么Redis是单线程且效率极高：

- 1) .绝大部分请求是纯粹的内存操作。
- 2) .避免了不必要的上下文切换和竞争条件，也不存在多进程或者多线程导致的切换而消耗 CPU，不用去考虑。各种锁的问题，不存在加锁释放锁操作。
- 3) .使用IO多路复用技术，可以处理并发的连接。



二、Redis数据类型

1 key键

1. **keys *** 查看当前库中所有的key 。

2. **exists key** 判断某个key是否存在。

可以设置多个key，只返回存在的个数，但不返回哪一个存在/不存在。

```
exists k1  查看k1是否存在，如果存在返回1
exists k1 k2 k3  查看k1 k2 k3是否存在，如果k1 k2存在，k3不存在，则返回2
```

3. **move key db** 将当前数据库的 key 移动到给定的数据库 db 当中。

```
move k1 8  将k1从当前数据库移动到8号库
```

4. **type key** 查看当前key 所储存的值的类型。

返回当前key所储存的值的类型，如string、list等。

5. **del key** 删除已存在的key，不存在的 key 会被忽略。

可以设置多个key，返回删除成功的个数。

```
del k1  删除k1，如果成功返回1，失败返回0
del k1 k2 k3  删除k1 k2 k3，如果k1 k2存在，k3不存在，则返回2
```

6. **expire key time** 给key设置time秒的过期时间。

设置成功返回 1 。当 key 不存在返回 0。

```
expire k1 10 给k1设置10秒后过期
```

7. **ttl key** 以秒为单位返回 key 的剩余过期时间。

当 key 不存在时，返回 -2 。当 key 存在但没有设置剩余生存时间时，返回 -1 。否则，以秒为单位，返回 key 的剩余生存时间。

8. **persist key** 移除给定 key 的过期时间，使得 key 永不过期。

当过期时间移除成功时，返回 1 。如果 key 不存在或 key 没有设置过期时间，返回 0 。

2 五大数据类型-String（字符串）

2.1 简介

String是Redis最基本的类型，一个key对应一个value。

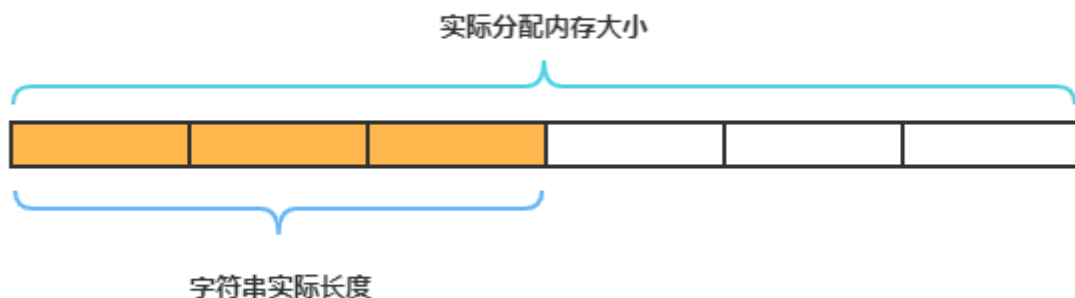
String是二进制安全的，意味着String可以包含任何数据，比如序列化对象或者一张图片。

String最多可以放512M的数据。

2.2 String底层数据结构

String底层数据结构是简单动态字符串（simple dynamic string,SDS）。

类似于 Python 中的 `List`，采用预分配方式来减少内存的频繁分配。



如图，内存实际大小一般都要高于字符串实际大小。当字符串长度小于1M时，扩容都是加倍现有的空间，如果超过1M，扩容时每次只会多扩1M的空间。字符串最大长度为512M。

2.3 常用命令

1. **set key value** 用于设置给定 key 的值。如果 key 已经存储其他值，set 就重写旧值，且无视类型。

```
set k1 v1 向Redis中设置一个k1的键值对
set k1 100 将k1的值由v1重置为100
```

2. **get key** 用于获取指定 key 的值。如果 key 不存在，返回 nil 。

3. **append key value** 将给定的value追加到key原值末尾。

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 就简单地给定 key 设为 value，就像执行 set key value 一样。

```
append k1 v1 在Redis中不存在k1，所以直接设置k1的值为v1
append k1 v2 向k1的值末尾添加一个v2，最终结果为v1v2
```

4. **strlen key** 获取指定 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。

5. **setex key time value** 给指定的 key 设置值及time 秒的过期时间。如果 key 已经存在， setex命令将会替换旧的值，并设置过期时间。

```
setex k1 10 v1 向Redis中设置一个k1的键值对并且10秒后过期
```

6. **setnx key value**当key不存在时，设置给定 key 的值。如果key存在，则没有任何影响。

```
setnx k1 v1 向Redis中设置一个k1的键值对
setnx k1 v2 Redis中存在k1，则没有影响，k1的值仍然为v1
```

7. **incr key** 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。

如字符串类型的值不能表示为数字、或者是其他类型，那么返回一个错误。

```
incr k1 因为Redis中不存在k1，所以先初始化为0，再递增，值为1
incr k1 存在k1，递增后k1的值为2
set k2 v2
incr k2 因为k2不为数值，Redis返回一个错误
```

8. **decr key**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。

如字符串类型的值不能表示为数字、或者是其他类型，那么返回一个错误。

```
decr k1 因为Redis中不存在k1，所以先初始化为0，再递增，值为-1
decr k1 存在k1，递增后k1的值为-2
set k2 v2
decr k2 因为k2不为数值，Redis返回一个错误
```

9. **incrby/decrby key step** 将key存储的数字值按照step进行增减。

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/decrby 命令。

如字符串类型的值不能表示为数字、或者是其他类型，那么返回一个错误。

10. **mset key1 value1 key2 value2**同时设置一个或多个 key-value。

```
mset k1 v1 k2 v2 k3 v3 同时向Redis中设置了k1 k2 k3
```

11. **mget key1 key2**返回所有(一个或多个)给定 key 的值。

如果给定的 key 里面，有某个 key 不存在，那么这个 key 返回特殊值 nil。

```
mget k1 k2 同时获取k1 k2
```

12. **msetnx key1 value1 key2 value2**用于所有给定 key 都不存在时，同时设置一个或多个 key-value。

msetnx具有原子性特性，有一个失败，则都失败。

```
msetnx k1 v1 k2 v2 向Redis中设置k1 k2两个键值对
msetnx k1 v2 k3 v2 Redis中存在k1, k1设置失败，由于原子性特性，k3也设置失败
```

13. **getrange key start end**用于获取存储在指定 key 中字符串的子字符串。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)。

```
set java helloworld 设置一个key为java, value为helloworld的值
getrange java 0 3 获取索引0-3的值，结果为hell
```

14. **setrange key offset value**用指定的字符串重写给定 key 所储存的字符串值，重写的位置从偏移量 offset 开始。

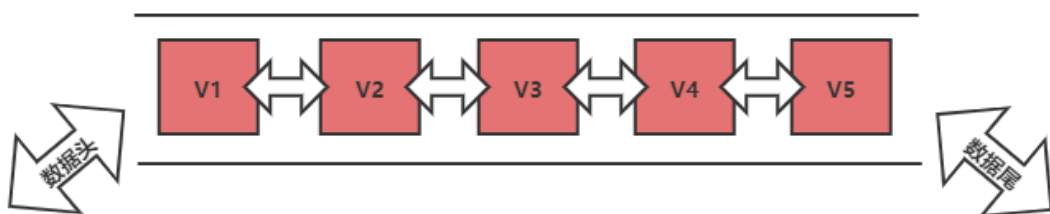
```
set java helloworld 设置一个key为java, value为helloworld的值
setrange java 5 baizhan 从偏移位置5 (w) 开始，用baizhan重写key
```

3 五大数据类型-List（列表）

3.1 简介

List是简单的字符串列表，按照插入顺序排序。你可以添加一个元素到列表的头部（左边）或者尾部（右边）。

底层是一个双向链表，对两端操作性能极高，通过索引操作中间的节点性能较差。

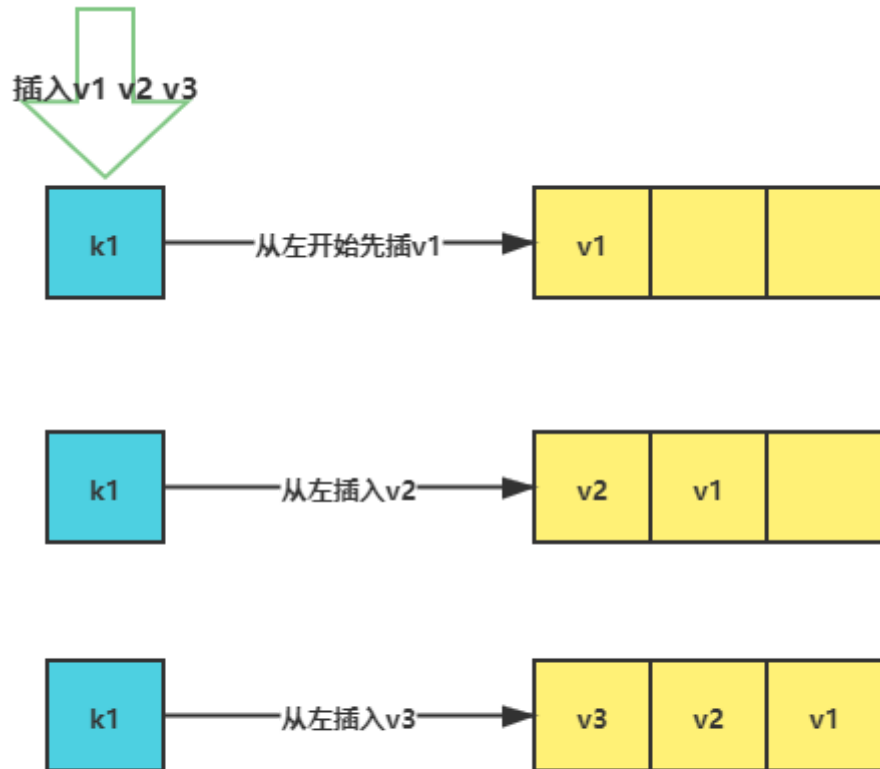


一个List最多可以包含 $2^{32}-1$ 个元素（每个列表超过40亿个元素）。

3.2 常用命令

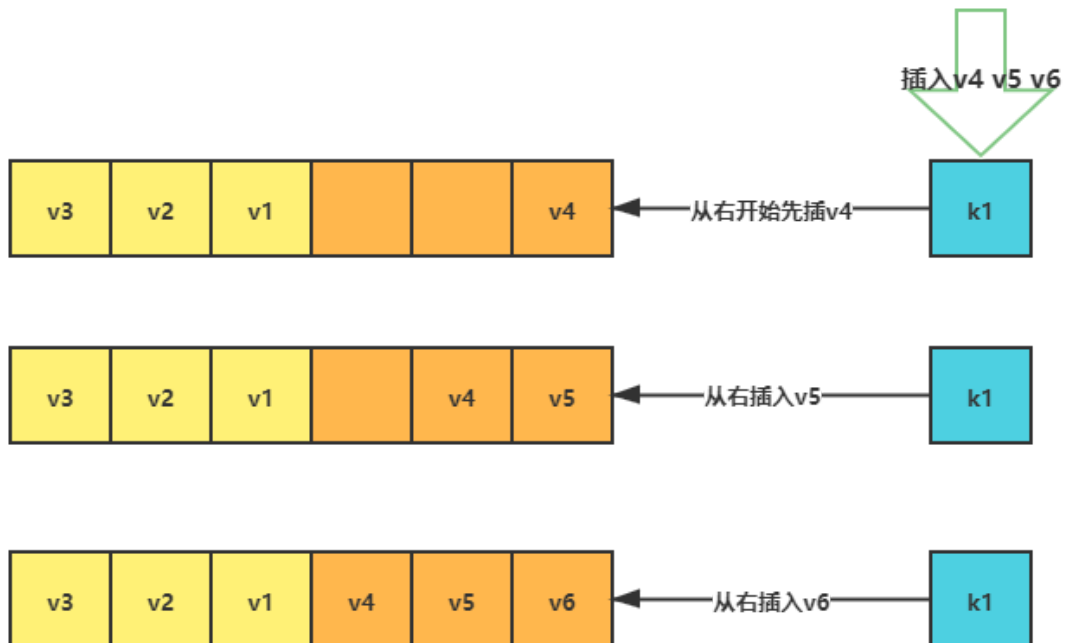
1. **lpush/ rpush key1 value1 value2 value3.....**从左边（头部）/右边（尾部）插入一个或多个值。

```
lpush k1 v1 v2 v3 从左边放入v1 v2 v3
```



lpush过程

rpush k1 v4 v5 v6 从右边放入v4 v5 v6



rpush过程

2. **lrange key start end**返回key列表中的start和end之间的元素（包含start和end）。其中 0 表示列表的第一个元素，-1表示最后一个元素。

```
lrange k1 0 2 取出列表里前3个值，结果为v3 v2 v1
lrange k1 0 -1 取出列表里全部值，结果为v3 v2 v1 v4 v5 v6
```

3. lpop/rpop key 移除并返回第一个值/最后一个值。

值在键在，值光键亡。

```
lpop k1 从列表中删除v3，并返回，当前列表全部值v2 v1 v4 v5 v6
rpop k1 从列表中删除v6，并返回，当前列表全部值v2 v1 v4 v5
```

4. lindex key index 获取列表index位置的值（从左开始）。

5. llen key 获取列表长度。

6. lrem key count value 从左边开始删除与value相同的count个元素。

```
lrem k1 2 v1 从左边开始删除k1列表中2个v1元素
```

7. linsert key before/after value newvalue 在列表中value值的前边/后边插入一个新value值（从左开始）。

```
linsert k1 before v1 v5 在v1前面插入一个v5
```

8. lset key index value 将索引为index的值设置为value

4 五大数据类型-Set（集合）

4.1 简介

与List类似是一个列表功能，但Set是自动排重的，当需要存储一个列表数据，又不希望出现重复数据时，Set是一个很好的选择。

Set是String类型的无序集合，它底层其实是一个value为null的hash表，所以添加、删除、查找的时间复杂度都是O(1)。

一般来说，一个算法如果是O(1)，随着数据增加，查找数据的时间不变。

集合中最大的成员数为 $2^{32}-1$ （每个集合超过40亿个元素）。

4.2 常用命令

1. sadd key value1 value2..... 将一个或多个元素添加到集合key中，已经存在的元素将被忽略。

```
sadd k1 v1 v2 v2 v3 v4 v5 v6 向集合中添加值，最终只有v1 v2 v3 v4 v5 v6
```

2. smembers key 取出该集合的所有元素。

```
smembers k1
```

3. srandmember key count 随机取出集合中count个元素，但不会删除。

```
srandmember k1 2 随机取出集合中的2个元素
```

4. sismember key value 判断集合key中是否含有value元素，如有返回1，否则返回0。

```
sismember k1 v1
```

5. **scard key**返回该集合的元素个数。

```
scard k1
```

6. **smove sourcekey destinationkey value**将value元素从sourcekey集合移动到destinationkey集合中。

如果 sourcekey集合不存在或不包含指定的 value元素，则 smove 命令不执行任何操作，仅返回 0。

```
smove k1 k2 v5 将元素v5从集合k1中移动到集合k2
```

7. **srem key value1 value2.....**删除集合中的一个或多个成员元素，不存在的成员元素会被忽略。

```
srem k1 v1 v2 删除v1 v2
```

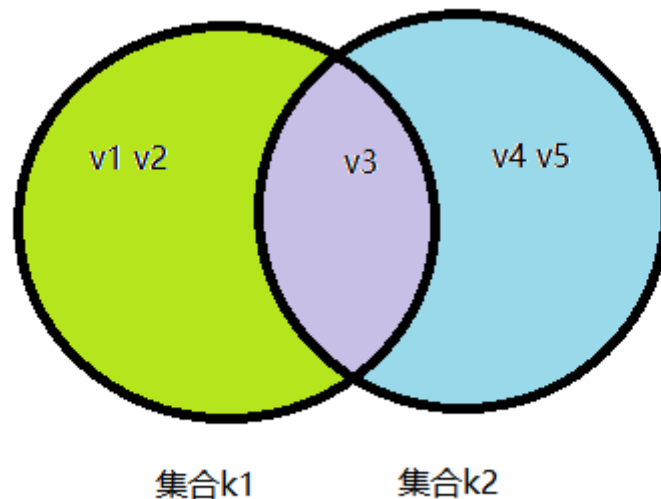
8. **spop key**随机删除集合中一个元素并返回该元素。

```
spop k1 随机删除一个元素，并返回
```

9. **sinter key1 key2**返回两个集合的交集元素。

10. **sunion key1 key2**返回两个集合的并集元素。

11. **sdiff key1 key2**返回两个集合的差集元素（key1中的，不包含key2）



```
sinter k1 k2 返回v3  
sunion k1 k2 返回v1 v2 v3 v4 v5  
sdiff k1 k2 返回v1 v2  
sdiff k2 k1 返回v4 v5
```

5 五大数据类型-Zset（有序集合）

5.1 简介

Zset与Set非常相似，是一个没有重复元素的String集合。

不同之处是Zset的每个元素都关联了一个分数（score），这个分数被用来按照从低分到高分的方式排序集合中的元素。集合的元素是唯一的，但分数可以重复。

因为元素是有序的，所以可以根据分数（score）或者次序（position）来获取一个范围内的元素。

5.2 常用命令

1. **zadd key score1 value1 score2 value2.....** 将一个或多个元素（value）及分数（score）加入到有序集key中。

如果某个元素已经是有序集的元素，那么更新这个元素的分数值，并通过重新插入这个元素，来保证该元素在正确的位置上。

分数值可以是整数值或双精度浮点数。

如果有序集合 key 不存在，则创建一个空的有序集并执行 zadd 操作。

```
zadd k1 100 python 200 js 300 sql 400 css
```

2. **zrange key start end [withscores]** 返回key集合中的索引start和索引end之间的元素（包含start和end）。

其中元素的位置按分数值递增(从小到大)来排序。其中 0 表示列表的第一个元素，-1表示最后一个元素。

withscores是可选参数，是否返回分数。

```
zrange k1 0 -1 返回集合中所有元素  
zrange k1 0 -1 withscores 返回集合中所有元素，并携带元素分数
```

3. **zrangebyscore key minscore maxscore [withscores]** 返回key集合中的分数minscore 和分数maxscore 之间的元素（包含minscore 和maxscore）。其中元素的位置按分数值递增(从小到大)来排序。

```
zrangebyscore k1 200 400 返回200-400分之间的元素递增排序
```

4. **zrevrangebyscore key maxscore minscore [withscores]** 返回key集合中的分数maxscore和分数minscore 之间的元素（包含maxscore和minscore）。其中元素的位置按分数值递减(从大到小)来排序。

```
zrevrangebyscore k1 400 200 返回200-400分之间的元素递减法排序
```

5. **zincrby key increment value**为元素value的score加上increment的值。

```
zincrby k1 50 java 给java元素加上50分
```

6. **zcount key minscore maxscore**统计该集合在minscore 到maxscore分数区间中元素的个数。

```
zcount k1 100 300 统计100分到300分中间元素的个数
```

7. **zrank key value**返回value在集合中的排名，从0开始。


```
zrank k1 sql 返回sql排名
```

8. **zrem key value**删除该集合下value的元素。

```
zrem k1 css 删除css
```

6 五大数据类型-Hash（哈希）

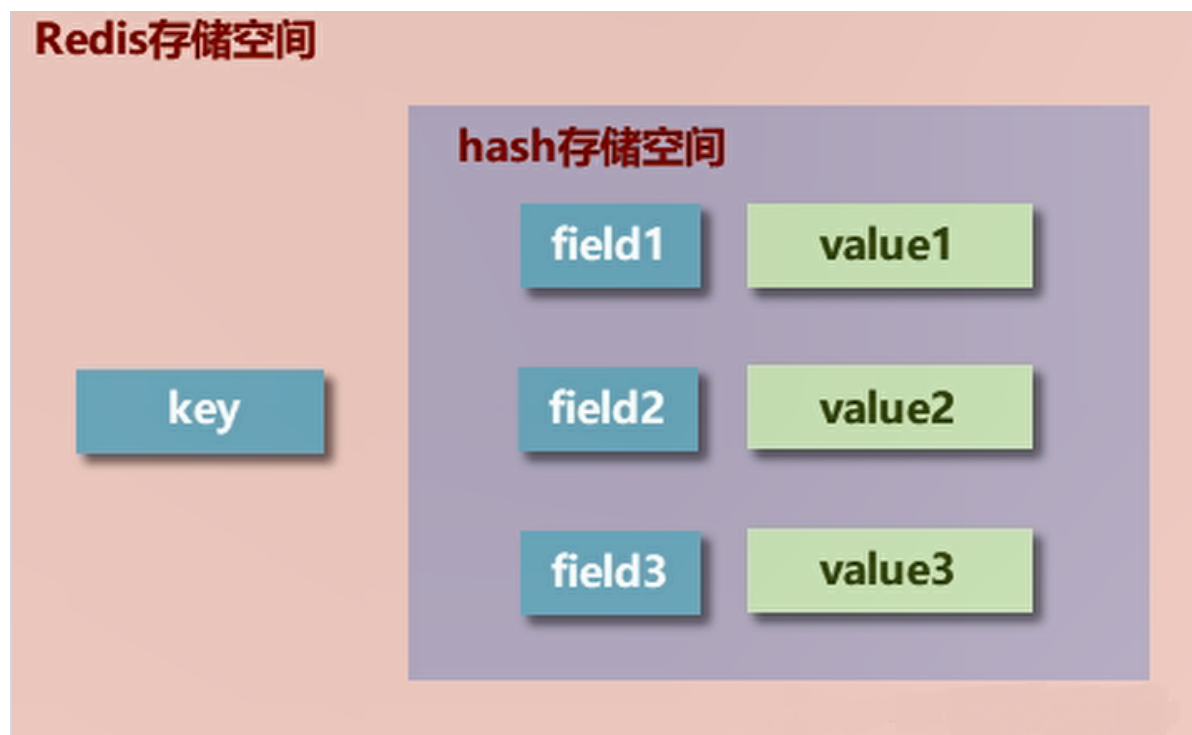
6.1 简介

Hash是一个键值对的集合。

Hash 是一个 String 类型的 field（字段）和 value（值）的映射表，hash 特别适合用于存储对象。

Hash 是 Redis 中出现最为频繁的复合型数据结构,除了 dict 结构的数据会用到Hash外,整个 Redis 数据库的所有 key 和 value 也组成了一个全局Hash,还有带过期时间的 key 集合也是一个Hash。set集合相当于一个value为null的Hash，zset 集合中存储 value 和 score 值的映射关系也是通过 hash 结构实现的。

每个 Hash 可以存储 $2^{32} - 1$ 键值对（40多亿）



6.2 常用命令

1. **hset key field value**给key集合中的field赋值value。

如果哈希表不存在，一个新的哈希表被创建并进行 HSET 操作。

如果字段已经存在于哈希表中，旧值将被重写。

```
hset user name baizhan 创建一个key为用户的哈希，并创建name字段，给name字段赋值为baizhan
hset user age 3          在key为用户的哈希中，创建age字段赋值为3
hset user name shangxuetang 将key为用户的哈希name字段修改为shangxuetang
```

2. **hget key field**从key哈希中，取出field字段的值。

```
hget user name 从key为用户的哈希中取出name字段的值，结果为shangxuetang
```

3. **hmset key field1 value1 field2 value2.....**批量设置哈希的字段及值。

```
hmset user1 name bjsxt age 15 创建一个key为用户1的哈希，有两个字段name和age
```

4. **hsetnx key field value** 给key哈希表中不存在的的字段赋值。

如果哈希表不存在，一个新的哈希表被创建并进行 hsetnx 操作。

如果字段已经存在于哈希表中，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。

```
hsetnx user name
```

5. **hexists key field** 判断指定key中是否存在field

如果哈希表含有给定字段，返回 1 。 如果哈希表不含有给定字段，或 key 不存在，返回 0 。

```
hexists user name 返回1
```

6. **hkeys key**获取该哈希中所有的field。

7. **hvals key**获取该哈希中所有的value。

8. **hincrby key field increment**为哈希表key中的field字段的值加上增量increment。

增量也可以为负数，相当于对指定字段进行减法操作。

如果哈希表的 key 不存在，一个新的哈希表被创建并执行 hincrby 命令。

如果指定的字段不存在，那么在执行命令前，字段的值被初始化为 0 。

对一个储存字符串值的字段执行 hincrby 命令将造成一个错误。

```
hincrby user age 10 对用户中的age字段做运算，增加10
```

9. **hdel key field1 field2.....**删除哈希表 key 中的一个或多个指定字段，不存在的字段将被忽略。

返回被成功删除字段的数量，不包括被忽略的字段。

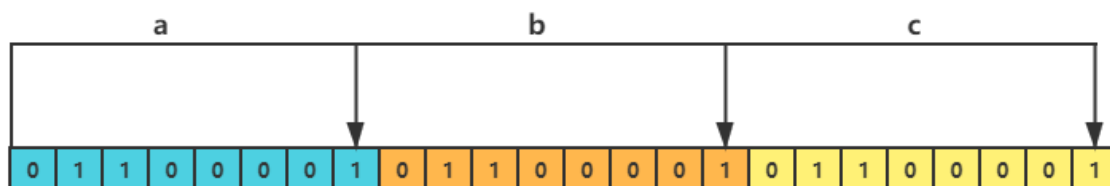
```
hdel user name age 删除user中的name和age字段
```

7 新数据类型-Bitmaps

7.1 简介

在计算机中，用二进制（位）作为存储信息的基本单位，1个字节等于8位。

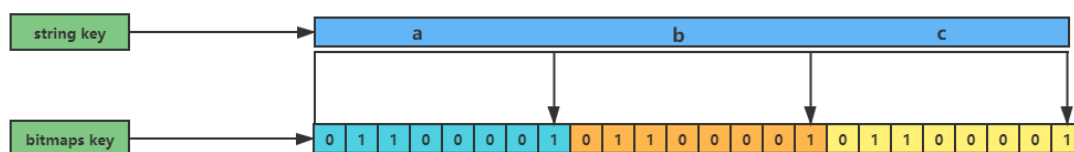
例如 "abc" 字符串是由 3 个字节组成，计算机存储时使用其二进制表示，"abc"分别对应的ASCII码是 97、98、99，对应的二进制是01100001、01100010、01100011，在内存中表示如下：



合理地使用 位 能够有效地提高内存使用率和开发效率。

Redis提供了Bitmaps这个“数据结构”可以实现对位的操作：

1. Bitmaps 本身不是一种数据结构，实际上它就是字符串（key 对应的 value 就是上图中的一串二进制），但是它可以对字符串的位进行操作。
2. Bitmaps 单独提供了一套命令，所以在 Redis 中使用 Bitmaps 和使用字符串的方法不太相同。可以把 Bitmaps 想象成一个以位为单位的数组，数组的每个单元只能存储 0 和 1，数组的下标在 Bitmaps 中叫做偏移量。



7.2 常用命令

1. **setbit key offset value** 设置Bitmaps中某个偏移量的值。

偏移量从0开始，且value值只能为0或1。

```
setbit sign 0 1 设置sign的第一位值为1
setbit sign 1 1 设置sign的第二位值为1
setbit sign 2 0 设置sign的第三位值为0
setbit sign 3 1 设置sign的第四位值为1
```



2. **getbit key offset** 获取Bitmaps中某个偏移量的值。

获取key的offset 的值。

```
getbit sign 3 获取偏移量为3的值，结果为1
```

如果偏移量未设置值，则也返回0。

```
getbit sign 99 获取偏移量为99的值，结果为0
```

3. **bitcount key [start end]**统计字符串被设置为1的bit数量。一般情况下，给定的整个字符串都会被进行统计，可以选择通过额外的start和end参数，指定**字节组**范围内进行统计（包括start和end），0表示第一个元素，-1表示最后一个元素。

```
bitcount sign 获取整个字符串被设置为1的bit数量，结果为3
```

如：当前存在一个key为k1的bitmaps存储着[00000001,00000001,00000010,00000011]，分别对应[1,1,2,3]。

```
setbit num 7 1
setbit num 15 1
setbit num 22 1
setbit num 30 1
setbit num 31 1
```

bitcount num 1 2 统计索引1、2两个字节组中bit=1的数量，即统计00000001,00000010中bit=1的数量，结果为2

bitcount num 1 3 统计索引1、2、3三个字节组中bit=1的数量，即统计00000001,00000010,00000011中bit=1的数量，结果为4

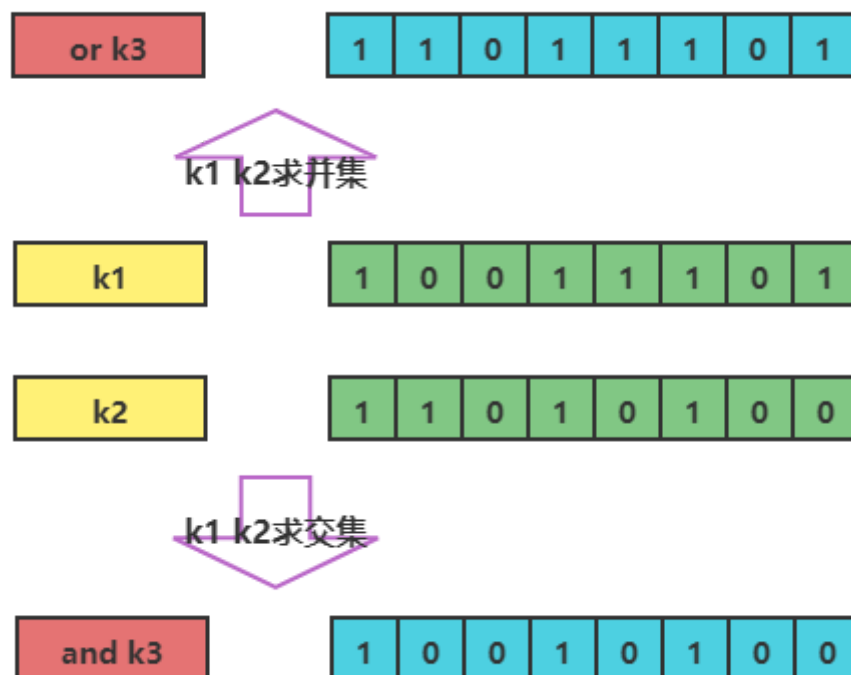
bitcount num 0 -1 统计所有的字节组中bit=1的数量，结果为5

setbit设置或获取的是bit（位）的位置，bitcount计算的是byte（字节）位置。

4. **bitop and/or destkey sourcekey1 sourcekey2.....**将多个bitmaps通过求交集/并集方式合并成一个新的bitmaps。

bitop and k3 k1 k2 通过求交集将k1 k2合并成k3

bitop or k3 k1 k2 通过求并集将k1 k2合并成k3



8 新数据类型-Geospatia

8.1 简介

GEO, Geographic,地理信息的缩写。

该类型就是元素的二维坐标，在地图上就是经纬度。Redis基于该类型，提供了经纬度设置、查询、范围查询、距离查询、经纬度Hash等常见操作。

8.2 常用命令

1. **geoadd key longitude latitude member [longitude latitude member]**用于存储指定的地理空间位置，可以将一个或多个经度(longitude)、纬度(latitude)、位置名称(member)添加到指定的 key 中。

```
geoadd chinacity 116.405285 39.904989 beijing 将北京的经纬度和名称添加到chinacity
geoadd chinacity 104.065735 30.659462 chengdu 121.472644 31.231706 shanghai
将成都和上海的经纬度、名称添加到chinacity
```

有效的经度：-180 ~ +180 有效的纬度：-85.05 ~ +85.05，当设置的经度纬度值超过范围会报错。

两级无法直接添加。

一般会直接下载城市数据，直接通过 Python 程序直接一次性导入。

2. **geopos key member [member]****从给定的 key 里返回所有指定名称(member)的位置（经度和纬度），不存在的返回 nil。

```
geopos chinacity shanghai beijing 返回chinacity中名称为shanghai和beijing的经纬度
```

3. **geodist key member1 member2 [m|km|ft|mi]**用于返回两个给定位置之间的距离。

最后一个距离单位参数说明：

- o m：米，默认单位。
- o km：千米。
- o mi：英里。
- o ft：英尺。

```
geodist chinacity shanghai beijing 返回shanghai和beijing之间的距离，结果
1067597.9668，单位米
geodist chinacity shanghai chengdu km 返回shanghai和chengdu之间的距离，结果
1660.0198，单位是千米
```

4. **georadius key longitude latitude radius m|km|ft|mi** 以给定的经纬度（longitude latitude）为中心，返回键包含的位置元素当中，与中心的距离不超过给定最大距离（radius）的所有位置元素。

```
georadius chinacity 116 40 1200 km 获取经纬度116 40为中心，在chinacity内1200公里
范围内的所有元素。结果shanghai beijing
```

9 新数据类型-Hyperloglog

9.1 简介

在我们做站点流量统计的时候一般会统计页面UV(独立访客:unique visitor)和PV(即页面浏览量: page view)。

什么是基数?

数据集 {1, 2, 5, 7, 5, 7, 9} , 那么这个数据集的基数集为 {1, 2, 5, 7, 9} , 基数 (不重复元素) 为5, 基数估计就是在误差可接受范围内, 快速计算基数。

如果是通过Redis来处理, 我们可以使用String类型然后自增计数即可达到统计PV, 统计UV可以使用Set, 每个用户id是唯一的可以放到这个集合里。

以上方案虽然结果准确, 但随着数据不断增加, 导致占用的内存空间越来越大, 对于非常大的数据集是不合适的。

Hyperloglog 是一种基数估算统计, 在输入元素的数量特别巨大时, 计算基数所需的空间是固定的, 并且很小。

在Redis中, 每个Hyperloglog 只占用12KB内存, 就可以计算接近 2^{64} 个不同元素的基数。

因为HyperLogLog 只会更具输入元素来计算基数, 而不会存储输入元素本身, 所以Hyperloglog 不能像集合那样, 返回输入的各个元素。

9.2 常用命令

1. **pfadd key element1 element2.....**将所有元素参数添加到 Hyperloglog 数据结构中。

如果至少有个元素被添加返回 1, 否则返回 0。

```
pfadd book1 python sql 添加两个元素, 当前book1数量为2
pfadd book1 python js 添加一个元素, 当前book1数量为3
```

2. **pfcount key1 key2.....**计算Hyperloglog 近似基数, 可以计算多个Hyperloglog , 统计基数总数。

```
pfcount book1 计算book1的基数, 结果为3
pfadd book2 html css 添加两个元素到book2中
pfcount book1 book2 统计两个key的基数总数, 结果为5
```

3. **pfmerge destkey sourcekey1 sourcekey2.....**将一个或多个Hyperloglog (sourcekey1) 合并成一个Hyperloglog (destkey) 。

比如每月活跃用户可用每天活跃用户合并后计算。

```
pfmerge book book1 book2 将book1和book2合并成book, 结果为5
```

10 Redis配置文件详解

10.1 units单位

配置大小单位, 开头定义基本度量单位, 只支持bytes, 大小写不敏感。

```

8 # Note on units: when memory size is needed, it is possible to specify
9 # it in the usual form of 1k 5GB 4M and so forth:
10 #
11 # 1k => 1000 bytes
12 # 1kb => 1024 bytes
13 # 1m => 1000000 bytes
14 # 1mb => 1024*1024 bytes
15 # 1g => 1000000000 bytes
16 # 1gb => 1024*1024*1024 bytes
17 #
18 # units are case insensitive so 1GB 1Gb 1gB are all the same.
19

```

10.2 INCLUDES

Redis只有一个配置文件，如果多个人进行开发维护，那么就需要多个这样的配置文件，这时候多个配置文件就可以在此通过 `include /path/to/local.conf` 配置进来，而原本的 `redis.conf` 配置文件就作为一个总闸。

```

20 ##### INCLUDES #####
21
22 # Include one or more other config files here. This is useful if you
23 # have a standard template that goes to all Redis servers but also need
24 # to customize a few per-server settings. Include files can include
25 # other files, so use this wisely.
26 #
27 # Note that option "include" won't be rewritten by command "CONFIG REWRITE"
28 # from admin or Redis Sentinel. Since Redis always uses the last processed
29 # line as value of a configuration directive, you'd better put includes
30 # at the beginning of this file to avoid overwriting config change at runtime.
31 #
32 # If instead you are interested in using includes to override configuration
33 # options, it is better to use include as the last line.
34 #
35 # include /path/to/local.conf
36 # include /path/to/other.conf

```

10.3 NETWORK

1. `bind`：绑定redis服务器网卡IP，默认为127.0.0.1,即本地回环地址。访问redis服务只能通过本机的客户端连接，而无法通过远程连接。如果bind选项为空的话，那会接受所有来自于可用网络接口的连接。

```

72 # IF YOU ARE SURE YOU WANT YOUR INSTANCE TO LISTEN TO ALL THE INTERFACES
73 # JUST COMMENT OUT THE FOLLOWING LINE.
74 # ~~~~~
75 bind 127.0.0.1 -:::1

```

2. `protected-mode`：本机保护模式，值为yes时只能本机访问不能远程访问。

```

90 # By default protected mode is enabled. You should disable it only if
91 # you are sure you want clients from other hosts to connect to Redis
92 # even if no authentication is configured, nor a specific set of interfaces
93 # are explicitly listed using the "bind" directive.
94 protected-mode yes

```

3. `port`：指定redis运行的端口，默认是6379。

```

96 # Accept connections on the specified port, default is 6379 (IANA #815344).
97 # If port 0 is specified Redis will not listen on a TCP socket.
98 port 6379

```

4. `timeout`：设置客户端连接时的超时时间，单位为秒。当客户端在这段时间内没有发出任何指令，那么关闭该连接。默认值为0，表示不关闭。

```

118 # Close the connection after a client is idle for N seconds (0 to disable)
119 timeout 0

```


10.4 GENERAL

1. daemonize: 设置为yes表示指定Redis以守护进程的方式启动（后台启动）。

```
254 # By default Redis does not run as a daemon. Use 'yes' if you need it.
255 # Note that Redis will write a pid file in /var/run/redis.pid when daemonized.
256 # When Redis is supervised by upstart or systemd, this parameter has no impact.
257 daemonize yes
```

2. pidfile: 配置PID文件路径, 当redis作为守护进程运行的时候, 它会把 pid 默认写到 /var/redis/run/redis_6379.pid 文件里面。

```
287 # Note that on modern Linux systems "/run/redis.pid" is more conforming
288 # and should be used instead.
289 pidfile /var/run/redis 6379.pid
```

3. loglevel: 定义日志级别。默认值为notice, 有如下4种取值:

- debug (记录大量日志信息, 适用于开发、测试阶段)。
- verbose (较多日志信息)。
- notice (适量日志信息, 使用于生产环境)。
- warning (仅有部分重要、关键信息才会被记录)。

```
291 # Specify the server verbosity level.
292 # This can be one of:
293 # debug (a lot of information, useful for development/testing)
294 # verbose (many rarely useful info, but not a mess like the debug level)
295 # notice (moderately verbose, what you want in production probably)
296 # warning (only very important / critical messages are logged)
297 loglevel notice
```

4. logfile: 配置log文件地址, 默认打印在命令行终端的窗口上。

```
299 # Specify the log file name. Also the empty string can be used to force
300 # Redis to log on the standard output. Note that if you use standard
301 # output for logging but daemonize, logs will be sent to /dev/null
302 logfile ""
```

5. databases: 设置数据库的数目。

```
324 # Set the number of databases. The default database is DB 0, you can select
325 # a different one on a per-connection basis using SELECT <dbid> where
326 # dbid is a number between 0 and 'databases'-1
327 databases 16
```

10.5 SECURITY

1. requirepass: 设置redis连接密码。

比如: requirepass 123 表示redis的连接密码为123。

```
898 # The requirepass is not compatible with aclfile option and the ACL LOAD
899 # command, these will cause requirepass to be ignored.
900 #
901 # requirepass foobared
```

10.6 其他配置

1. maxclients: 设置客户端最大并发连接数, 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数量。如果设置 maxclients为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错误信息。

```
960 # IMPORTANT: When Redis Cluster is used, the max number of connections is also
961 # shared with the cluster bus: every node in the cluster will use two
962 # connections, one incoming and another outgoing. It is important to size the
963 # limit accordingly in case of very large clusters.
964 #
965 # maxclients 10000
```


2. maxmemory: 设置Redis的最大内存, 如果设置为0。表示不作限制。通常是配合maxmemory-policy参数一起使用。

```
988 # In short... if you have replicas attached it is suggested that you set a lower
989 # limit for maxmemory so that there is some free RAM on the system for replica
990 # output buffers (but this is not needed if the policy is 'noeviction').
991 #
992 # maxmemory <bytes>
```

3. maxmemory-policy: 当内存使用达到maxmemory设置的最大值时, redis使用的内存清除策略。

清除策略包括:

- volatile-lru: 利用LRU算法移除设置过过期时间的key (LRU:最近使用 Least Recently Used)
- allkeys-lru: 利用LRU算法移除任何key
- volatile-random: 移除设置过过期时间的随机key
- allkeys-random: 移除随机key
- volatile-ttl: 移除即将过期的key(minor TTL)
- noeviction: 不移除任何key, 只是返回一个写错误, 默认选项

```
994 # MAXMEMORY POLICY: how Redis will select what to remove when maxmemory
995 # is reached. You can select one from the following behaviors:
996 #
997 # volatile-lru -> Evict using approximated LRU, only keys with an expire set.
998 # allkeys-lru -> Evict any key using approximated LRU.
999 # volatile-lfu -> Evict using approximated LFU, only keys with an expire set.
1000 # allkeys-lfu -> Evict any key using approximated LFU.
1001 # volatile-random -> Remove a random key having an expire set.
1002 # allkeys-random -> Remove a random key, any key.
1003 # volatile-ttl -> Remove the key with the nearest expire time (minor TTL)
1004 # noeviction -> Don't evict anything, just return an error on write operations.
1005 #
1006 # LRU means Least Recently Used
1007 # LFU means Least Frequently Used
1008 #
1009 # Both LRU, LFU and volatile-ttl are implemented using approximated
1010 # randomized algorithms.
1011 #
1012 # Note: with any of the above policies, when there are no suitable keys for
1013 # eviction, Redis will return an error on write operations that require
1014 # more memory. These are usually commands that create new keys, add data or
1015 # modify existing keys. A few examples are: SET, INCR, HSET, LPUSH, SUNIONSTORE,
1016 # SORT (due to the STORE argument), and EXEC (if the transaction includes any
1017 # command that requires memory).
1018 #
1019 # The default is:
1020 #
1021 # maxmemory-policy noeviction
1022
```

单位

#当你需要为某个配置项指定内存大小的时候, 必须要带上单位,

#通常的格式就是 1k 5gb 4m 等:

#1k => 1000 bytes

#1kb => 1024 bytes

#1m => 1000000 bytes

#1mb => 10241024 bytes

#1g => 1000000000 bytes

#1gb => 10241024*1024 bytes

文件引入

#引入其他的配置文件

#include /path/to/local.conf

#include /path/to/other.conf

模块加载

#启动时加载模块

#loadmodule /path/to/my_module.so

#loadmodule /path/to/other_module.so

网络

#指定redis只能接受来自此IP绑定的网卡请求，注意此默认值默认外网是不可访问的

bind 127.0.0.1

#是否开启保护模式。如果没有指定bind和密码，redis只会本地进行访问，拒绝外部访问。

protected-mode yes

#默认端口，建议生产环境不要使用默认端口避免被恶意扫描到

port 6379

#TCP连接中已完成队列(完成三次握手之后)的长度

tcp-backlog 511

#配置unix socket来让redis支持监听本地连接。

#unixsocket /tmp/redis.sock

#配置unix socket使用文件的权限

#unixsocketperm 700

#客户端连接空闲超过timeout将会被断开，为0则断开

timeout 0

#tcp keepalive参数

tcp-keepalive 300

基本配置

#是否后台启动

daemonize no

#可以通过upstart和systemd管理Redis守护进程

#选项:

#supervised no - 没有监督互动

#supervised upstart - 通过将Redis置于SIGSTOP模式来启动信号

#supervised systemd - signal systemd将READY = 1写入\$ NOTIFY_SOCKET

#supervised auto - 检测upstart或systemd方法基于 UPSTART_JOB或NOTIFY_SOCKET环境变量

supervised no

#配置PID文件路径

pidfile /var/run/redis_6379.pid

#日志级别

#参数:

debug

verbose

notice

warning

loglevel notice

#日志文件

logfile ""

#是否打开记录syslog功能

#syslog-enabled no

```
#syslog标识符
#syslog-ident redis

#日志的来源
#syslog-facility local0

#数据库的数量，默认使用的数据库是DB 0
#可以通过”SELECT “命令选择一个db
#集群环境默认只有DB 0
databases 16

#是否一直显示logo
always-show-logo yes

数据持久化RDB

#保存数据到磁盘：
#save

#Will save the DB if both the given number of seconds and the given
#number of write operations against the DB occurred.

#In the example below the behaviour will be to save:

#15分钟有一个key发生变化就保存数据到磁盘
#after 900 sec (15 min) if at least 1 key changed

#5分钟有10个key发生变化就保存数据到磁盘
#after 300 sec (5 min) if at least 10 keys changed

#1分钟有10000个key发生变化就保存数据到磁盘
#after 60 sec if at least 10000 keys changed

#Note: you can disable saving completely by commenting out all “save” lines.

#还可以删除所有以前配置的保存。
#通过添加带有单个空字符串参数的保存指令
#like in the following example:

save 900 1
save 300 10
save 60 10000

#持久化出现错误后，是否依然进行继续进行工作
stop-writes-on-bgsave-error yes

#是否校验rdb文件
rdbcompression yes

#使用压缩rdb文件，rdb文件压缩使用LZF压缩算法，
rdbchecksum yes

#rdb文件名称
dbfilename dump.rdb

#rdb使用上面的“dbfilename配置指令的文件名保存到这个目录
dir ./
```

主从复制

#指定主节点。旧版本是：slaveof

#replicaof

#master的密码

#masterauth

#当一个slave失去和master的连接，或者同步正在进行中，slave的行为有两种可能：

#如果 replica-serve-stale-data 设置为“yes”（默认值），slave会继续响应客户端请求，可能是正常数据，也可能是还没获得值的空数据。

#如果 replica-serve-stale-data 设置为“no”，slave会回复"正在从master同步（SYNC with master in progress）"来处理各种请求，除了 INFO 和 SLAVEOF 命令。

replica-serve-stale-data yes

#配置从是否为只读，开启后从则不能写入数据，旧版本是：slave-read-only yes

replica-read-only yes

#同步策略：磁盘或socket，默认磁盘方式

repl-diskless-sync no

#如果非磁盘同步方式开启，可以配置同步延迟时间，以等待master产生子进程通过socket传输RDB数据给slave。

#默认值为5秒，设置为0秒则每次传输无延迟。

repl-diskless-sync-delay 5

#slave根据指定的时间间隔向master发送ping请求。默认10秒。

#repl-ping-replica-period 10

#同步的超时时间

#slave在与master SYNC期间有大量数据传输，造成超时

#在slave角度，master超时，包括数据、ping等

#在master角度，slave超时，当master发送REPLCONF ACK pings#确保这个值大于指定的repl-ping-slave-period，否则在主从间流量不高时每次都会检测到超时

#repl-timeout 60

#是否在slave套接字发送SYNC之后禁用 TCP_NODELAY

#如果选择yes，Redis将使用更少的TCP包和带宽来向slaves发送数据。但是这将使数据传输到slave上有延迟，Linux内核的默认配置会达到40毫秒。

#如果选择no，数据传输到slave的延迟将会减少但要使用更多的带宽。

#默认我们会为低延迟做优化，但高流量情况或主从之间的跳数过多时，可以设置为“yes”。

repl-disable-tcp-nodelay no

#设置数据备份的backlog大小

#repl-backlog-size 1mb

#从最后一个slave断开开始计时多少秒后，backlog缓冲将会释放。

#repl-backlog-ttl 3600

#优先级

replica-priority 100

#如果master少于N个延时小于等于M秒的已连接slave，就可以停止接收写操作。

#N个slave需要是“online”状态。

#延时是以秒为单位，并且必须小于等于指定值，是从最后一个从slave接收到的ping（通常每秒发送）开始计数。

#该选项不保证N个slave正确同步写操作，但是限制数据丢失的窗口期。

#例如至少需要3个延时小于等于10秒的slave用下面的指令：

#min-replicas-to-write 3

#min-replicas-max-lag 10

安全

#密码

#requirepass foobared

#命令重命名

#设置命令为空时禁用命令

#rename-command CONFIG ""

限制

#设置最多同时连接的客户端数量

#maxclients 10000

#内存限制

#maxmemory

#如果达到上方最大的内存限制，Redis如何选择删除key

#volatile-lru -> 根据LRU算法删除设置过期时间的key

#allkeys-lru -> 根据LRU算法删除任何key

#volatile-random -> 随机移除设置过过期时间的key

#allkeys-random -> 随机移除任何key

#volatile-ttl -> 移除即将过期的key(minor TTL)

#noeviction -> 不移除任何key，只返回一个写错误

#注意：对所有策略来说，如果Redis找不到合适的可以删除的key都会在写操作时返回一个错误。

#目前为止涉及的命令：set setnx setex append incr decr rpush lpush rpushx lpushx

linsert lset rpoplpush sadd sinter sinterstore sunion sunionstore sdiff

sdiffstore zadd zincrby zunionstore zinterstore hset hsetnx hmset hincrby incrby

decrby getset mset msetnx exec sort

#maxmemory-policy noeviction

#LRU和最小TTL算法的样本个数

#maxmemory-samples 5

懒删除

#内存满逐出

lazyfree-lazy-eviction no

#过期key删除

lazyfree-lazy-expire no

#内部删除，比如rename oldkey newkey时，如果newkey存在需要删除newkey

lazyfree-lazy-server-del no

#接收完RDB文件后清空数据选项

replica-lazy-flush no

持久化方式AOF

#每次启动时Redis都会先把这个文件的数据读入内存里，先忽略RDB文件

appendonly no

#AOF文件名称

appendfilename "appendonly.aof"

#fsync() 系统调用告诉操作系统把数据写到磁盘上，而不是等更多的数据进入输出缓冲区。

#有些操作系统会真的把数据马上刷到磁盘上；有些则会尽快去尝试这么做。

#Redis支持三种不同的模式：

#no: 不要立刻刷, 只有在操作系统需要刷的时候再刷。比较快。
#always: 每次写操作都立刻写入到aof文件。慢, 但是最安全。
#everysec: 每秒写一次。折中方案。
#默认的“everysec”通常来说能在速度和数据安全性之间取得比较好的平衡。
appendfsync everysec

#如果AOF的同步策略设置成“always”或者“everysec”, 并且后台的存储进程(后台存储或写入AOF日志)会产生很多磁盘I/O开销。某些Linux的配置下会使Redis因为 fsync()系统调用而阻塞很久。
#注意, 目前对这个情况还没有完美修正, 甚至不同线程的 fsync() 会阻塞我们同步的write(2)调用。
#为了缓解这个问题, 可以用下面这个选项。它可以在 BGSAVE 或 BGREWRITEAOF 处理时阻止fsync()。
#这就意味着如果有子进程在进行保存操作, 那么Redis就处于"不可同步"的状态。
#这实际上是说, 在最差的情况下可能会丢掉30秒钟的日志数据。(默认Linux设定)
#如果把这个设置成"yes"带来了延迟问题, 就保持"no", 这是保存持久数据的最安全的方式。
no-appendfsync-on-rewrite no

#自动重写AOF文件。如果AOF日志文件增大到指定百分比, Redis能够通过 BGREWRITEAOF 自动重写AOF日志文件。
#工作原理: Redis记住上次重写时AOF文件的大小(如果重启后还没有写操作, 就直接用启动时的AOF大小)
#这个基准大小和当前大小做比较。如果当前大小超过指定比例, 就会触发重写操作。
#你还需要指定被重写日志的最小尺寸, 这样避免了达到指定百分比但尺寸仍然很小的情况还要重写。
#指定百分比为0会禁用AOF自动重写特性。

auto-aof-rewrite-percentage 100
#文件达到大小阈值的时候进行重写
auto-aof-rewrite-min-size 64mb

#如果设置为yes, 如果一个因异常被截断的AOF文件被redis启动时加载进内存, redis将会发送日志通知用户
#如果设置为no, redis将会拒绝启动。此时需要用"redis-check-aof"工具修复文件。

aof-load-truncated yes

#加载时Redis识别出AOF文件以“REDIS”开头字符串,
#并加载带此前缀的RDB文件, 然后继续加载AOF
aof-use-rdb-preamble yes

Lua脚本配置

#Lua 脚本的最大执行毫秒数

lua-time-limit 5000

集群

#开启redis集群

#cluster-enabled yes

#配置redis自动生成的集群配置文件名。确保同一系统中运行的各redis实例该配置文件不要重名。
#cluster-config-file nodes-6379.conf

#集群节点超时毫秒数

#cluster-node-timeout 15000

#如果数据太旧, 集群中的不可用master的slave节点会避免成为备用master。如果slave和master失联时间超过:(node-timeout * slave-validity-factor) + repl-ping-slave-period则不会被提升为master。
#如node-timeout为30秒, slave-validity-factor为10, 默认default repl-ping-slave-period为10秒, 失联时间超过310秒slave就不会成为master。
#较大的slave-validity-factor值可能允许包含过旧数据的slave成为master, 同时较小的值可能会阻止集群选举出新master。

#为了达到最大限度的高可用性，可以设置为0，即slave不管和master失联多久都可以提升为master
#cluster-replica-validity-factor 10

#只有在之前master有其它指定数量的工作状态下的slave节点时，slave节点才能提升为master。默认为1（即该集群至少有3个节点，1 master+2 slaves，master宕机，仍有另外1个slave的情况下其中1个slave可以提升）

#测试环境可设置为0，生成环境中至少设置为1

#cluster-migration-barrier 1

#默认情况下如果redis集群如果检测到至少有1个hash slot不可用，集群将停止查询数据。

#如果所有slot恢复则集群自动恢复。

#如果需要集群部分可用情况下仍可提供查询服务，设置为no。

#cluster-require-full-coverage yes

#选项设置为yes时，会阻止replicas尝试对其master在主故障期间进行故障转移

#然而，master仍然可以执行手动故障转移,如果强制这样做的话。

#cluster-replica-no-failover no

Docker集群配置

#默认情况下，Redis会自动检测自己的IP和从配置中获取绑定的PORT，告诉客户端或者是其他节点。

#而在Docker环境中，如果使用的不是host网络模式，在容器内部的IP和PORT都是隔离的，那么客户端和其他节点无法通过节点公布的IP和PORT建立连接。

#如果开启以下配置，Redis节点会将配置中的这些IP和PORT告知客户端或其他节点。而这些IP和PORT是通过Docker转发到容器内的临时IP和PORT的。

#cluster-announce-ip

#cluster-announce-port

#集群总线端口

#cluster-announce-bus-port

慢查询日志

#记录超过多少微秒的查询命令

#1000000等于1秒，设置为0则记录所有命令

slowlog-log-slower-than 10000

#记录大小，可通过SLOWLOG RESET命令重置

slowlog-max-len 128

延时监控系统

#记录执行时间大于或等于预定时间（毫秒）的操作,为0时不记录

latency-monitor-threshold 0

事件通知

#Redis能通知 Pub/Sub 客户端关于键空间发生的事件，默认关闭

notify-keyspace-events ""

内部数据结构

#当hash只有少量的entry时，并且最大的entry所占空间没有超过指定的限制时，会用一种节省内存的

#数据结构来编码。可以通过下面的指令来设定限制

hash-max-ziplist-entries 512

hash-max-ziplist-value 64

#当取正值的时候，表示按照数据项个数来限定每个quicklist节点上的ziplist长度。比如，当这个参数配置

#成5的时候，表示每个quicklist节点的ziplist最多包含5个数据项。

#当取负值的时候，表示按照占用字节数来限定每个quicklist节点上的ziplist长度。这时，它只能取-1到-5

#这五个值，每个值含义如下：

#-5: 每个quicklist节点上的ziplist大小不能超过64 Kb。（注: 1kb => 1024 bytes）
#-4: 每个quicklist节点上的ziplist大小不能超过32 Kb。
#-3: 每个quicklist节点上的ziplist大小不能超过16 Kb。
#-2: 每个quicklist节点上的ziplist大小不能超过8 Kb。（-2是Redis给出的默认值）
#-1: 每个quicklist节点上的ziplist大小不能超过4 Kb。
list-max-ziplist-size -2

#这个参数表示一个quicklist两端不被压缩的节点个数。

#注: 这里的节点个数是指quicklist双向链表的节点个数, 而不是指ziplist里面的数据项个数。

#实际上, 一个quicklist节点上的ziplist, 如果被压缩, 就是整体被压缩的。

#参数list-compress-depth的取值含义如下:

#0: 是个特殊值, 表示都不压缩。这是Redis的默认值。

#1: 表示quicklist两端各有1个节点不压缩, 中间的节点压缩。

#2: 表示quicklist两端各有2个节点不压缩, 中间的节点压缩。

#3: 表示quicklist两端各有3个节点不压缩, 中间的节点压缩。

#依此类推...

#由于0是个特殊值, 很容易看出quicklist的头节点和尾节点总是不被压缩的, 以便于在表的两端进行快速存取。

list-compress-depth 0

#set有一种特殊编码的情况: 当set数据全是十进制64位有符号整型数字构成的字符串时。

#下面这个配置项就是用来设置set使用这种编码来节省内存的最大长度。

set-max-intset-entries 512

#与hash和list相似, 有序集合也可以用一种特别的编码方式来节省大量空间。

#这种编码只适合长度和元素都小于下面限制的有序集合

zset-max-ziplist-entries 128

zset-max-ziplist-value 64

#HyperLogLog稀疏结构表示字节的限制。该限制包括

#16个字节的头。当HyperLogLog使用稀疏结构表示

#这些限制, 它会被转换成密度表示。

#值大于16000是完全没用的, 因为在该点

#密集表示是更多的内存效率。

#建议值是3000左右, 以便具有的内存好处, 减少内存的消耗

hll-sparse-max-bytes 3000

#Streams宏节点最大大小/项目。流数据结构是基数编码内部多个项目的大节点树。使用此配置

#可以配置单个节点的字节数, 以及切换到新节点之前可能包含的最大项目数

#追加新的流条目。如果以下任何设置设置为0, 忽略限制, 因此例如可以设置一个

#大入口限制将max-bytes设置为0, 将max-entries设置为所需的值

stream-node-max-bytes 4096

stream-node-max-entries 100

#启用哈希刷新, 每100个CPU毫秒会拿出1个毫秒来刷新Redis的主哈希表(顶级键值映射表)

activeremhashing yes

#客户端的输出缓冲区的限制, 可用于强制断开那些因为某种原因从服务器读取数据的速度不够快的客户端

client-output-buffer-limit normal 0 0 0

client-output-buffer-limit slave 256mb 64mb 60

client-output-buffer-limit pubsub 32mb 8mb 60

#客户端查询缓冲区累积新命令。它们仅限于固定的默认情况下,

#多数情况下为了避免协议不同步导致客户端查询缓冲区中未绑定的内存使用量的错误

#但是, 如果你有使用的话, 你可以在这里配置它, 比如我们有很多执行请求或类似的。

client-query-buffer-limit 1gb

#在Redis协议中, 批量请求, 即表示单个的元素strings, 通常限制为512 MB。


```
#但是，您可以z更改此限制
#proto-max-bulk-len 512mb

#默认情况下，“hz”的被设定为10。提高该值将在Redis空闲时使用更多的CPU时，但同时当有多个key
#同时到期会使Redis的反应更灵敏，以及超时可以更精确地处理
hz 10

#开启动态hz
dynamic-hz yes

#当一个子进程重写AOF文件时，如果启用下面的选项，则文件每生成32M数据会被同步
aof-rewrite-incremental-fsync yes

#当redis保存RDB文件时，如果启用了以下选项，每生成32 MB数据，文件将被fsync-ed。
#这很有用，以便以递增方式将文件提交到磁盘并避免大延迟峰值。
rdb-save-incremental-fsync yes

碎片整理
#启用主动碎片整理
#activedefrag yes

#启动活动碎片整理的最小碎片浪费量
#active-defrag-ignore-bytes 100mb

#启动碎片整理的最小碎片百分比
#active-defrag-threshold-lower 10

#使用最大消耗时的最大碎片百分比
#active-defrag-threshold-upper 100

#在CPU百分比中进行碎片整理的最小消耗
#active-defrag-cycle-min 5

#磁盘碎片整理的最大消耗
#active-defrag-cycle-max 75

#将从主字典扫描处理的最大set / hash / zset / list字段数
#active-defrag-max-scan-fields 1000
```

11 发布与订阅

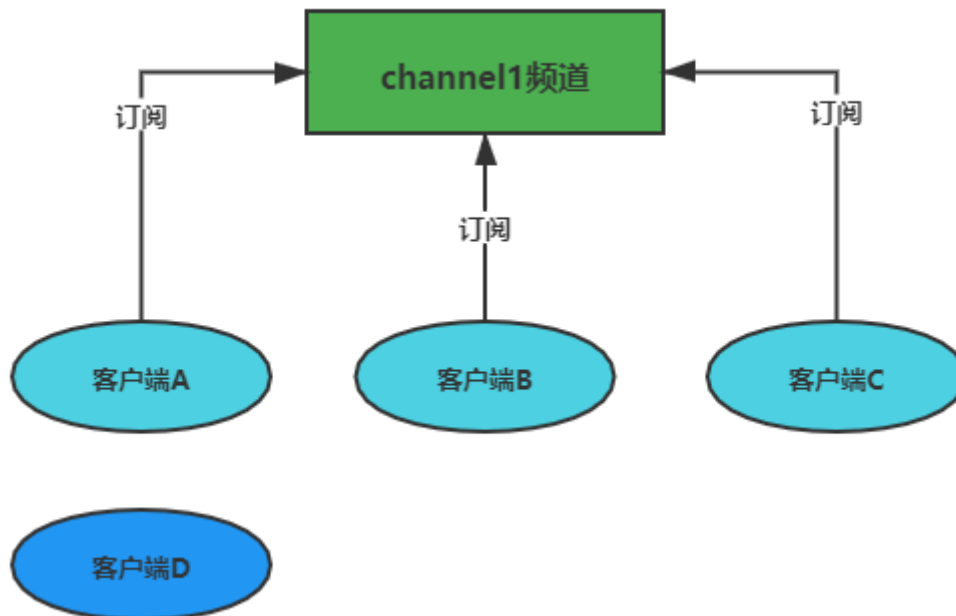
11.1 什么是发布与订阅

Redis 发布订阅 (pub/sub) 是一种消息通信模式：发送者 (pub) 发送消息，订阅者 (sub) 接收消息。

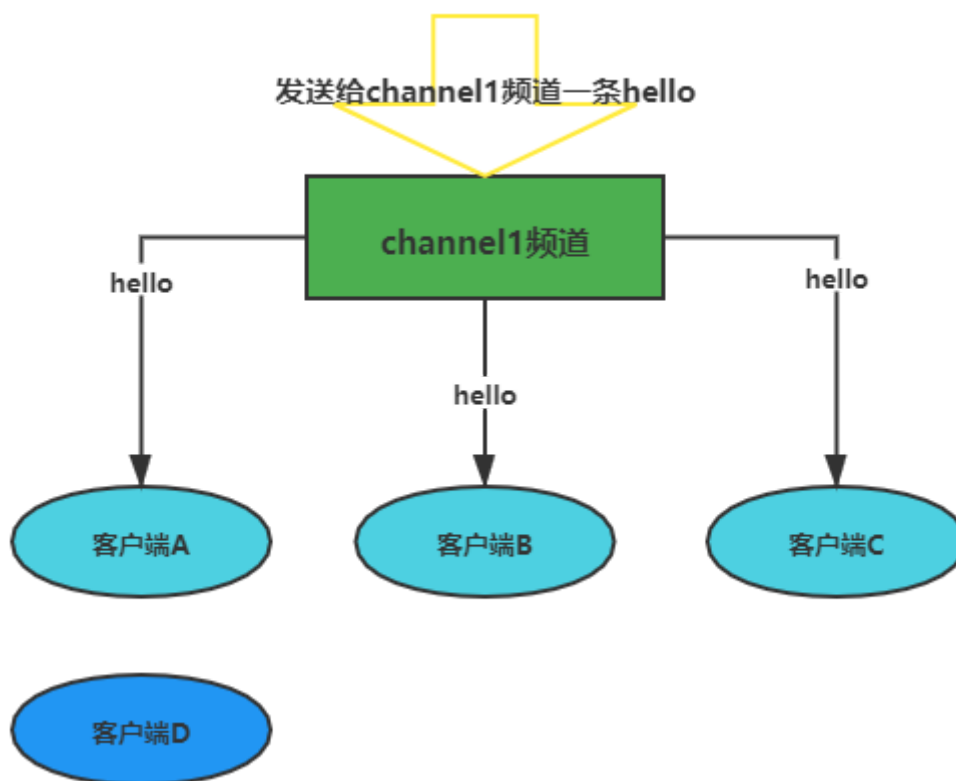
Redis 客户端可以订阅任意数量的频道。

11.2 Redis的发布与订阅

1. 客户端订阅频道



2. 当给这个频道发送消息后，消息就会发送给订阅的客户端



3. Redis中发布与订阅命令

订阅：subscribe channel 订阅频道channel。

发布：publish channel msg向频道channel 发送一条msg消息。

11.3 发布与订阅命令行实现

1. 打开一个客户端订阅channel 1频道。

```
127.0.0.1:6379> subscribe channel1
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "channel1"
3) (integer) 1
█
```

2. 打开另一个客户端给channel 1频道发送一条hello消息。

```
127.0.0.1:6379> publish channel1 hello
(integer) 1
127.0.0.1:6379> █
```

返回1代表订阅者数量。

3. 打开第一个客户端可以看到发送的消息

```
127.0.0.1:6379> subscribe channel1
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "channel1"
3) (integer) 1
1) "message"
2) "channel1"
3) "hello"
█
```

客户端只能收到订阅以后发送的消息。

三、Redis持久化

由于Redis的数据都存放在内存中，如果没有配置持久化，Redis重启后数据就全丢失了，于是需要开启Redis的持久化功能，将数据保存到磁盘上，当Redis重启后，可以从磁盘中恢复数据。

Redis提供了两个不同形式的持久化方式：

- RDB(Redis DataBase)
- AOF(Append Only File)

1 持久化操作-RDB

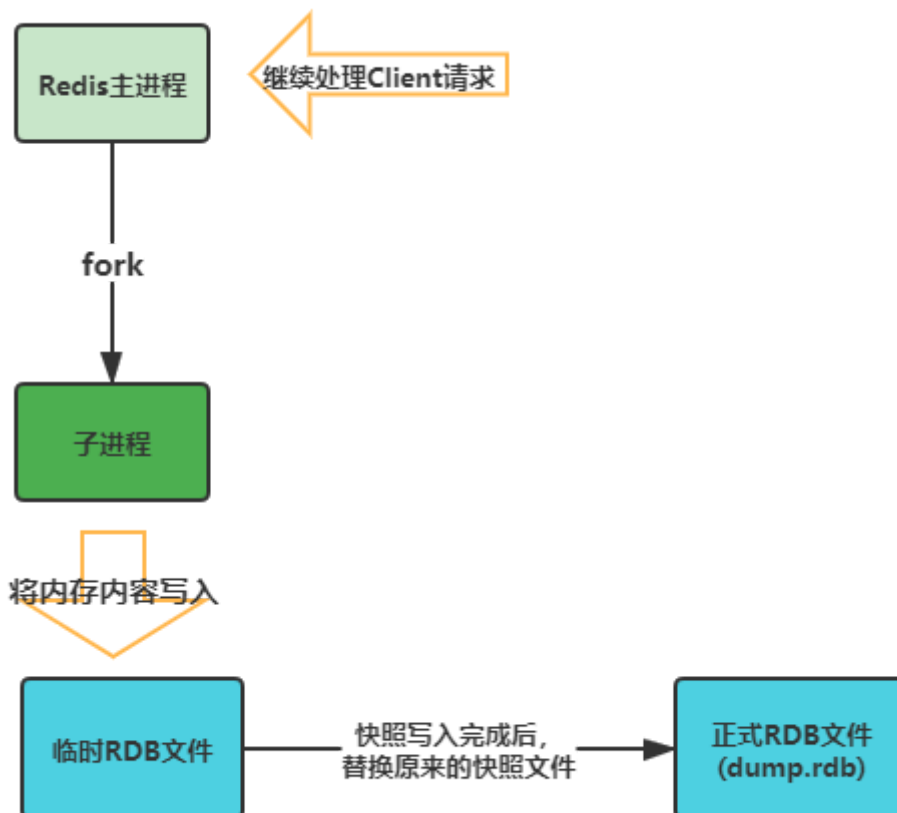
1.1 RDB是什么？

在指定的时间间隔内将内存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。

1.2 备份过程

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。



1.3 dump.rdb文件

1. RDB保存的文件，在redis.conf中配置文件名称，默认为dump.rdb。

```
429  
430 # The filename where to dump the DB  
431 dbfilename dump.rdb  
432
```

2. rdb文件的保存位置，也可以修改。默认在Redis启动时命令行所在的目录下。

```
[root@localhost bin]# pwd  
/usr/local/bin  
[root@localhost bin]# ls  
dump.rdb  redis-benchmark  redis-check-aof  redis-check-rdb  redis-cli  redis-sentinel  redis-server  
[root@localhost bin]#
```

redis.conf中配置文件路径

```
448 # The DB will be written inside this directory, with the filename specified  
449 # above using the 'dbfilename' configuration directive.  
450 #  
451 # The Append Only File will also be created inside this directory.  
452 #  
453 # Note that you must specify a directory here, not a file name.  
454 dir ./  
455
```

1.4 如何触发快照？

1.4.1 配置文件中默认的快照配置

```

360 ##### SNAPSHOTTING #####
361 █
362 # Save the DB to disk.
363 #
364 # save <seconds> <changes>
365 #
366 # Redis will save the DB if both the given number of seconds and the given
367 # number of write operations against the DB occurred.
368 #
369 # Snapshotting can be completely disabled with a single empty string argument
370 # as in following example:
371 #
372 # save ""
373 #
374 # Unless specified otherwise, by default Redis will save the DB:
375 # * After 3600 seconds (an hour) if at least 1 key changed
376 # * After 300 seconds (5 minutes) if at least 100 keys changed
377 # * After 60 seconds if at least 10000 keys changed
378 #
379 # You can set these explicitly by uncommenting the three following lines.
380 #
381 # save 3600 1
382 # save 300 100
383 # save 60 10000

```

1. 快照默认配置

save 3600 1: 表示3600秒内（一小时）如果至少有1个key的值变化，则保存
 save 300 100: 表示300秒内（五分钟）如果至少有100个 key 的值变化，则保存
 save 60 10000: 表示60秒内如果至少有 10000个key的值变化，则保存

可以自己配置新的保存规则。

- 例：给redis.conf添加新的快照策略，30秒内如果有5次key的变化，则触发快照。配置修改后，需要重启Redis服务。

```

380 #
381 # save 3600 1
382 # save 300 100
383 # save 60 10000
384
385 save 30 5
386

```

dump.rdb默认大小是92字节，里面会有一些基本信息。

```

[root@localhost bin]# ll
总用量 24844
-rw-r--r--. 1 root root 92 7月 5 22:30 dump.rdb
-rwxr-xr-x. 1 root root 6549080 6月 28 04:55 redis-benchmark
lrwxrwxrwx. 1 root root 12 6月 28 04:55 redis-check-aof -> redis-server
lrwxrwxrwx. 1 root root 12 6月 28 04:55 redis-check-rdb -> redis-server
-rwxr-xr-x. 1 root root 6763376 6月 28 04:55 redis-cli
lrwxrwxrwx. 1 root root 12 6月 28 04:55 redis-sentinel -> redis-server
-rwxr-xr-x. 1 root root 12118288 6月 28 04:55 redis-server
[root@localhost bin]# █

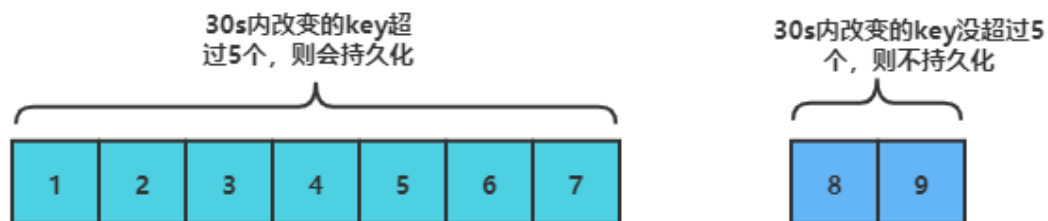
```

30秒内设置5个以上的值。

```
set k1 v1
set k2 v2
set k3 v3
set k4 v4
set k5 v5
set k6 v6
set k7 v7
```

```
[root@localhost bin]# ll
总用量 24844
-rw-r--r--. 1 root root    132 7月  5 22:32 dump.rdb
-rwxr-xr-x. 1 root root 6549080 6月 28 04:55 redis-benchmark
lrwxrwxrwx. 1 root root    12 6月 28 04:55 redis-check-aof -> redis-server
lrwxrwxrwx. 1 root root    12 6月 28 04:55 redis-check-rdb -> redis-server
-rwxr-xr-x. 1 root root 6763376 6月 28 04:55 redis-cli
lrwxrwxrwx. 1 root root    12 6月 28 04:55 redis-sentinel -> redis-server
-rwxr-xr-x. 1 root root 12118288 6月 28 04:55 redis-server
```

dump.rdb大小已经改变。



1.4.2 flushall

执行flushall命令, 也会触发rdb规则。

1.4.3 save与bgsave

手动触发Redis进行RDB持久化的命令有两种:

1. save

该命令会阻塞当前Redis服务器, 执行save命令期间, Redis不能处理其他命令, 直到RDB过程完成为止, 不建议使用。

save命令是同步命令, 会占用Redis的主进程。若Redis数据非常多时, save命令会执行速度非常慢, 阻塞所有客户端的请求

2. bgsave (推荐)

执行该命令时, Redis会在后台异步进行快照操作, 快照同时还可以响应客户端请求。

这两个命令是在Redis客户端中执行, 并不是redis.conf中修改

命令	save	bgsave
IO类型	同步	异步
阻塞	是	是(阻塞发生在fork(), 通常非常快)
复杂度	O(n)	O(n)
优点	不会消耗额外的内存	不阻塞客户端命令
缺点	阻塞客户端命令	需要fork子进程, 消耗内存

1.4.4 stop-writes-on-bgsave-error

默认值是yes。当Redis无法写入磁盘的话, 直接关闭Redis的写操作。

```
396 # However if you have setup your proper monitoring of the Redis server
397 # and persistence, you may want to disable this feature so that Redis will
398 # continue to work as usual even if there are problems with disk,
399 # permissions, and so forth.
400 stop-writes-on-bgsave-error yes
```

1.4.5 rdbcompression

默认值是yes。对于存储到磁盘中的快照, 可以设置是否进行压缩存储。如果是的话, redis会采用LZF算法进行压缩。如果你不想消耗CPU来进行压缩的话, 可以设置为关闭此功能, 但是存储在磁盘上的快照会比较大。

```
402 # Compress string objects using LZF when dump .rdb databases?
403 # By default compression is enabled as it's almost always a win.
404 # If you want to save some CPU in the saving child set it to 'no' but
405 # the dataset will likely be bigger if you have compressible values or keys.
406 rdbcompression yes
```

1.4.6 rdbchecksum

默认值是yes。在存储快照后, 我们还可以让redis使用CRC64算法来进行数据校验, 但是这样做会增加大约10%的性能消耗, 如果希望获取到最大的性能提升, 可以关闭此功能。

```
413 # RDB files created with checksum disabled have a checksum of zero that will
414 # tell the loading code to skip the check.
415 rdbchecksum yes
```

1.5 恢复数据

只需要将rdb文件放在Redis的启动目录, Redis启动时会自动加载dump.rdb并恢复数据

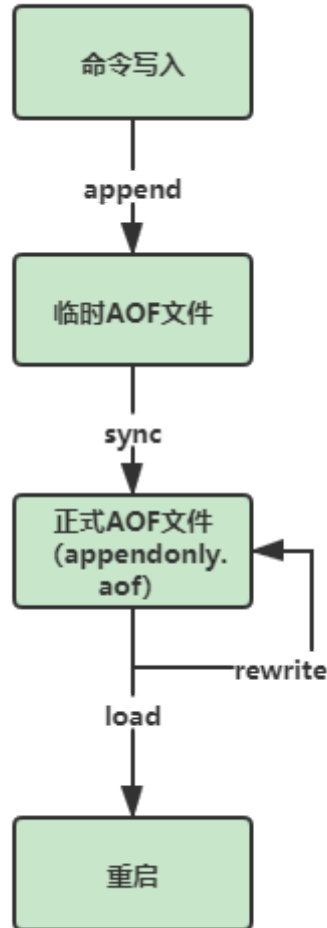
2 持久化操作-AOF

2.1 AOF是什么?

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), 只允许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。

2.2 AOF持久化流程

1. 客户端的请求写命令会被append追加到AOF缓冲区内。
2. AOF缓冲区根据AOF持久化策略[always,everysec,no]将操作同步到磁盘的AOF文件中。
3. AOF文件大小超过重写策略或手动重写时，会对AOF文件rewrite重写，压缩AOF文件容量。
4. Redis服务重启时，会重新load加载AOF文件中的写操作达到数据恢复的目的。



2.3 AOF默认不开启

可以在redis.conf中配置文件名称，默认为appendonly.aof。

AOF文件的保存路径，同RDB的路径一致。

```
# Please check https://redis.io/topics/persistence for more information.
appendonly no

# The name of the append only file (default: "appendonly.aof")
appendfilename "appendonly.aof"
```

如果AOF和RDB同时启动，Redis默认读取AOF的数据。

2.4 AOF启动/修复/恢复

- 正常恢复

1. 启动：设置Yes：修改默认的appendonly no，改为yes。
2. 恢复：重启Redis然后重新加载。

例：设置appendonly为yes，配置修改后，需要重启Redis服务。

```
1253  
1254 appendonly yes
```

服务器启动后，生成appendonly.aof文件，且大小为0。

```
[root@localhost bin]# ll  
总用量 24844  
-rw-r--r--. 1 root root      0 7月  6 01:33 appendonly.aof  
-rw-r--r--. 1 root root    139 7月  6 01:32 dump.rdb  
-rwxr-xr-x. 1 root root 6549080 6月 28 04:55 redis-benchmark  
lrwxrwxrwx. 1 root root    12 6月 28 04:55 redis-check-aof -> redis-server  
lrwxrwxrwx. 1 root root    12 6月 28 04:55 redis-check-rdb -> redis-server  
-rwxr-xr-x. 1 root root 6763376 6月 28 04:55 redis-cli  
lrwxrwxrwx. 1 root root    12 6月 28 04:55 redis-sentinel -> redis-server  
-rwxr-xr-x. 1 root root 12118288 6月 28 04:55 redis-server  
[root@localhost bin]#
```

设置数据。

```
set k11 v11  
set k12 v12  
set k13 v13  
set k14 v14  
set k15 v15
```

appendonly.aof大小已经改变。

```
[root@localhost bin]# ll  
总用量 24848  
-rw-r--r--. 1 root root    226 7月  6 01:37 appendonly.aof  
-rw-r--r--. 1 root root    132 7月  6 01:37 dump.rdb  
-rwxr-xr-x. 1 root root 6549080 6月 28 04:55 redis-benchmark  
lrwxrwxrwx. 1 root root    12 6月 28 04:55 redis-check-aof -> redis-server  
lrwxrwxrwx. 1 root root    12 6月 28 04:55 redis-check-rdb -> redis-server  
-rwxr-xr-x. 1 root root 6763376 6月 28 04:55 redis-cli  
lrwxrwxrwx. 1 root root    12 6月 28 04:55 redis-sentinel -> redis-server  
-rwxr-xr-x. 1 root root 12118288 6月 28 04:55 redis-server  
[root@localhost bin]#
```

- 异常恢复

1. 启动：设置Yes：修改默认的appendonly no，改为yes。
2. 修复：如遇到AOF文件损坏，通过/user/local/bin/redis-check-aof --fix appendonly.aof进行恢复。
3. 恢复：重启Redis然后重新加载。

例：服务启动和数据设置同上，模拟损坏appendonly.aof文件

```
vi appendonly.aof
```

在文件最后追加Hello World文本，破坏appendonly.aof原有格式，使其不可用。

```
k7
$2
v7
Hello World
```

重启Redis服务，并连接。由于aof文件被破坏，导致服务器启动失败。

```
[root@localhost bin]# ./redis-server ../myredis/redis.conf
[root@localhost bin]# ./redis-cli
Could not connect to Redis at 127.0.0.1:6379: Connection refused
not connected>
```

通过/user/local/bin/redis-check-aof --fix工具对appendonly.aof进行恢复。

```
./redis-check-aof --fix appendonly.aof
```

```
[root@localhost bin]# ./redis-check-aof --fix appendonly.aof
0x          e2: Expected prefix '*', got: 'H'
AOF analyzed: size=239, ok_up_to=226, ok_up_to_line=55, diff=13
This will shrink the AOF from 239 bytes, with 13 bytes, to 226 bytes
Continue? [y/N]: y
Successfully truncated AOF
```

修复成功。再次查看aof文件，破坏的地方已经修复。再次启动服务器成功。

```
k7
$2
v7
~
"appendonly.aof" [dos] 54L, 226C
```

2.5 AOF同步频率设置

```
1280
1281 # appendfsync always
1282 appendfsync everysec
1283 # appendfsync no
```

1. appendfsync always

始终同步，每次Redis的写入都会立刻记入日志，性能较差但数据完整性比较好。

2. appendfsync everysec

每秒同步，每秒记入日志一次，如果宕机，本秒的数据可能丢失。

3. appendfsync no

redis不主动进行同步，把同步时机交给操作系统。

2.6 Rewrite

1. 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。

例：设置k1为0，然后incr 进行了4次，k1对应的值会是4，其实就相当于set k1 4

2. 重写虽然可以节约大量磁盘空间，减少恢复时间。但是每次重写还是有一定的负担的，因此设定Redis要满足一定条件才会进行重写。

```
1322 # Specify a percentage of zero in order to disable the automatic AOF
1323 # rewrite feature.
1324
1325 auto-aof-rewrite-percentage 100
1326 auto-aof-rewrite-min-size 64mb
```

redis.conf默认配置

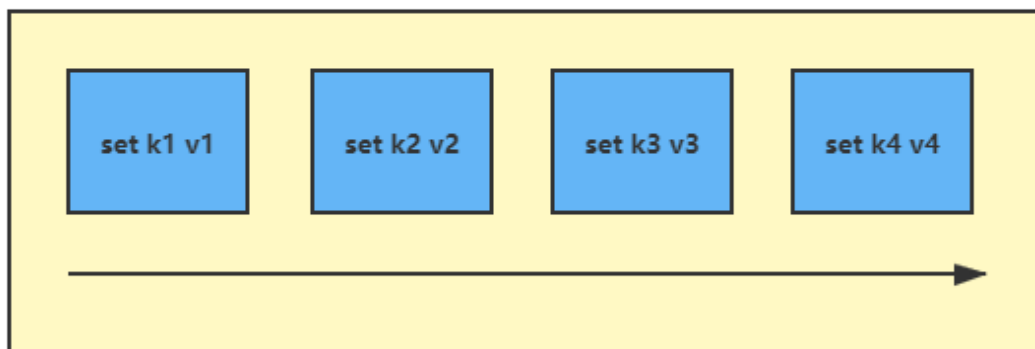
auto-aof-rewrite-min-size：表示重写时，文件大小必须比这个值要大。

auto-aof-rewrite-percentage：表示目前文件大小比上次重写后的文件大小大这么多才行。

四、Redis事务

1 Redis事务简介

1. Redis事务是一组命令的集合，一个事务中的所有命令都将被序列化，按照一次性、顺序性、排他性的执行队列系列的命令。



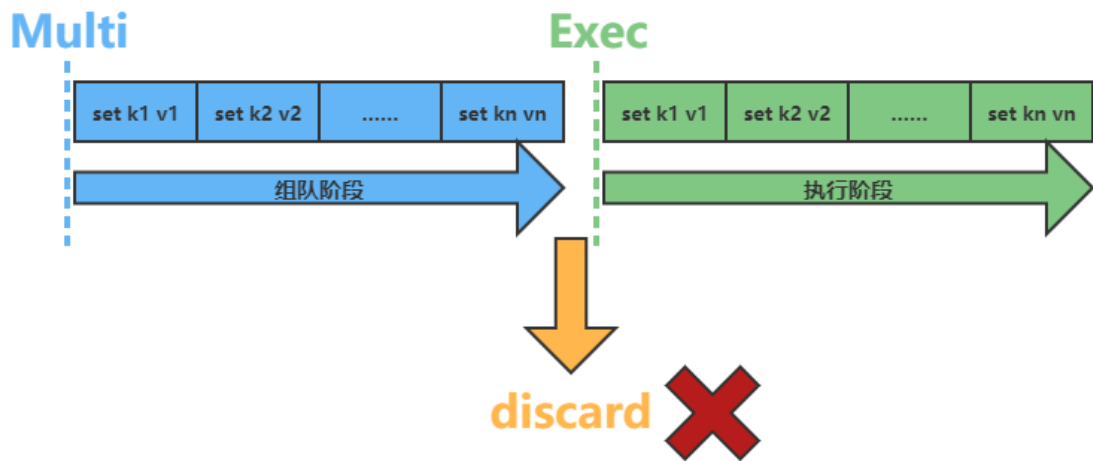
2. Redis单条命令保证原子性，但是事务不保证原子性，且没有回滚。事务中任意命令执行失败，其余的命令仍会被执行。
3. Redis事务没有隔离级别的概念。批量操作在执行前被放入缓存队列，并不会被实际执行，也就不存在事务内的查询要看到事务里的更新，事务外查询不能看到。
4. Redis事务的三个阶段：
 - 开始事务
 - 命令入队
 - 执行事务

2 Redis事务基本操作

2.1 Multi、Exec、discard

事务从输入Multi命令开始，输入的命令都会依次压入命令缓冲队列中，并不会执行，直到输入Exec后，Redis会将之前的命令缓冲队列中的命令依次执行。

组队过程中，可以通过discard来放弃组队。



例1:

```
multi    开始事务
set k1 v1 进行组队，并不执行
set k2 v2 进行组队，并不执行
exec     执行队列命令，依次设置k1 k2
```

```
127.0.0.1:6379> multi
OK
127.0.0.1:6379(TX)> set k1 v1
QUEUED
127.0.0.1:6379(TX)> set k2 v2
QUEUED
127.0.0.1:6379(TX)> exec
1) OK
2) OK
```

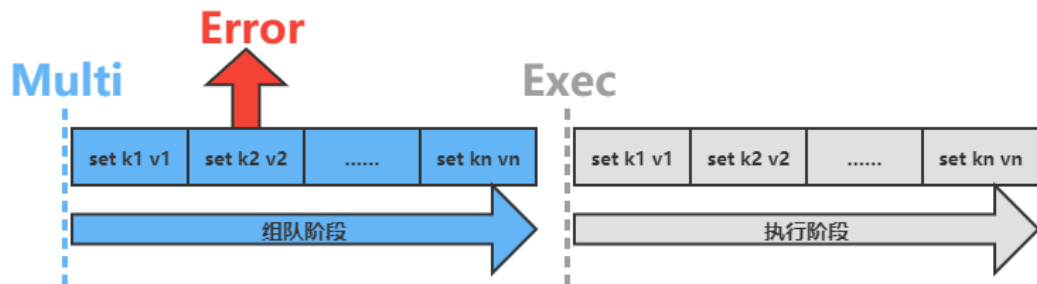
例2:

```
multi    开始事务
set k3 v3 进行组队，并不执行
set k4 v4 进行组队，并不执行
discard  取消组队，都不执行
```

```
127.0.0.1:6379> multi
OK
127.0.0.1:6379(TX)> set k3 v3
QUEUED
127.0.0.1:6379(TX)> set k4 v4
QUEUED
127.0.0.1:6379(TX)> discard
OK
127.0.0.1:6379> get k3
(nil)
127.0.0.1:6379>
```

2.2 事务的错误处理

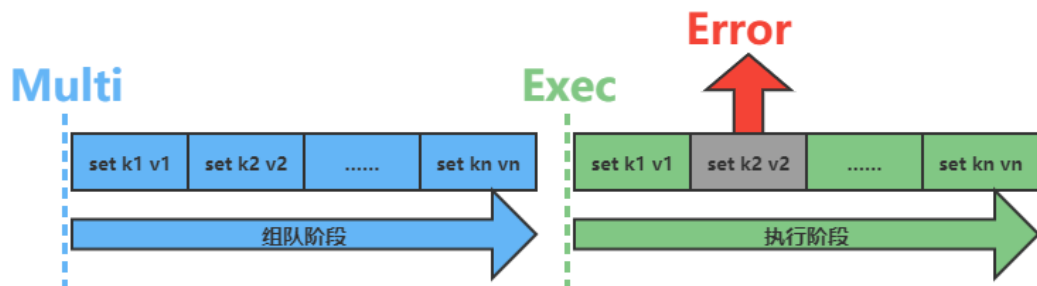
1. 语法错误，整个队列中的命令都不执行。



例：整个命令缓存队列都不会执行。

```
127.0.0.1:6379> multi
OK
127.0.0.1:6379(TX)> set b1 v1
QUEUED
127.0.0.1:6379(TX)> set b2 v2
QUEUED
127.0.0.1:6379(TX)> set b3
(error) ERR wrong number of arguments for 'set' command
127.0.0.1:6379(TX)> exec
(error) EXECABORT Transaction discarded because of previous errors.
127.0.0.1:6379> get b1
(nil)
127.0.0.1:6379>
```

2. 运行报错，只有报错的命令不会执行，其他正常执行。



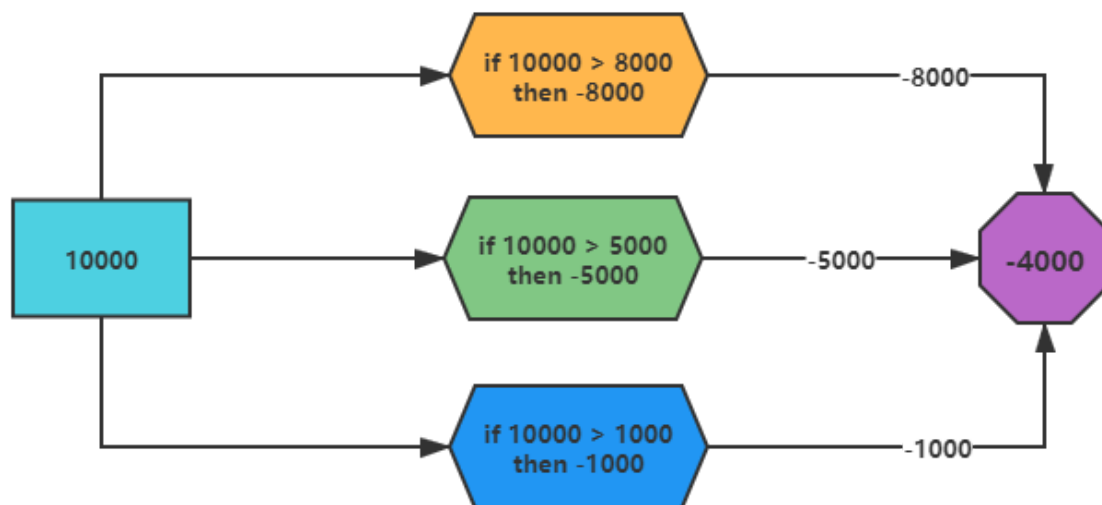
例：只有报错的命令没有执行。

```
127.0.0.1:6379> multi
OK
127.0.0.1:6379(TX)> set c1 v1
QUEUED
127.0.0.1:6379(TX)> incr c1
QUEUED
127.0.0.1:6379(TX)> set c2 v2
QUEUED
127.0.0.1:6379(TX)> exec
1) OK
2) (error) ERR value is not an integer or out of range
3) OK
127.0.0.1:6379> mget c1 c2
1) "v1"
2) "v2"
127.0.0.1:6379>
```

3 悲观锁与乐观锁

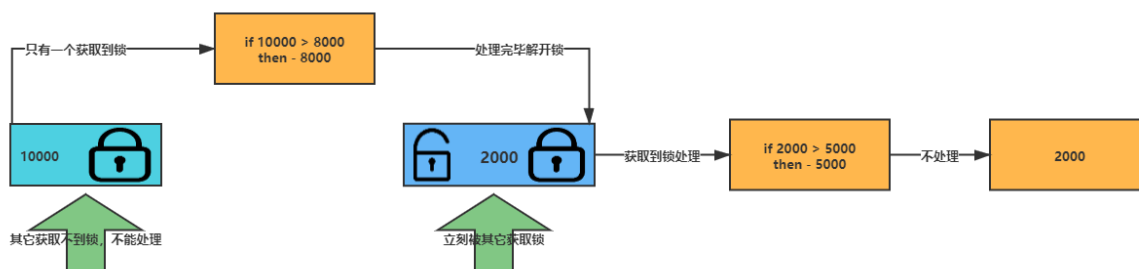
3.1 事务应用场景

- 一个请求想给余额减8000
- 一个请求想给余额减5000
- 一个请求想给余额减1000



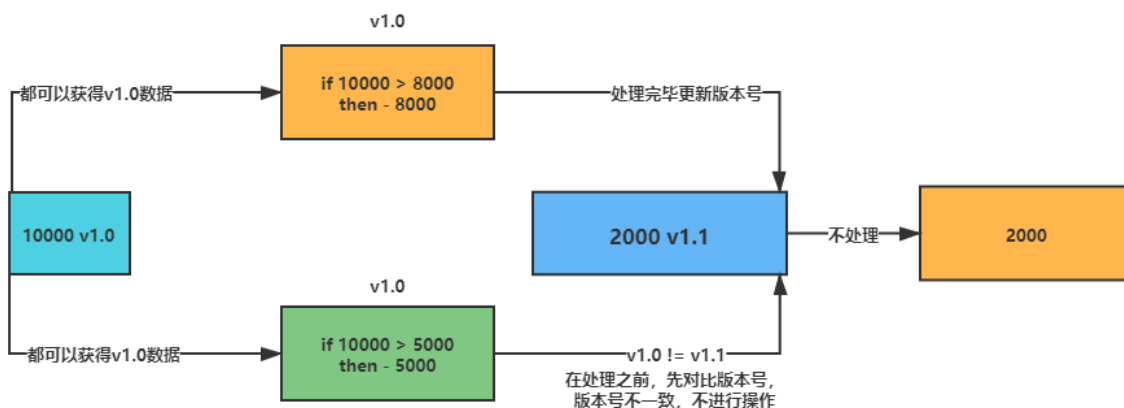
3.2 悲观锁 (Pessimistic Lock)

每次去拿数据的时候都认为别人会修改，所以在每次拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞，直到它拿到锁。传统关系型数据库就用到很多悲观锁，比如行锁、表锁等。



3.3 乐观锁 (Optimistic Lock)

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，可以使用版本号机制。乐观锁适用于多读的应用类型，可以提高吞吐量。Redis就是乐观锁机制实现事务的。



4 Redis中的乐观锁

4.1 watch key [key.....]

在执行multi之前，先执行watch监视一个或多个key，如果在事务执行之前这个（或这些）key被其它命令所改动，那么事务将被打断。

例：开启两个客户端

```
1号客户端
set money 10000 设置一个money的key
watch money 监视money
multi 开启事务
set money 100 修改money，暂时不执行事务

2号客户端
watch money 监视money
multi 开启事务
set money 110 修改money，暂时不执行事务

1号客户端
exec 执行事务，返回成功

2号客户端
exec 执行事务，更新失败
```

1号客户端

```
127.0.0.1:6379> set money 10000
OK
127.0.0.1:6379> watch money
OK
127.0.0.1:6379> multi
OK
127.0.0.1:6379(TX)> set money 100
QUEUED
127.0.0.1:6379(TX)> exec
1) OK
127.0.0.1:6379> █
```

2号客户端

```
127.0.0.1:6379> watch money
OK
127.0.0.1:6379> multi
OK
127.0.0.1:6379(TX)> set money 110
QUEUED
127.0.0.1:6379(TX)> exec
(nil)
127.0.0.1:6379> █
```

4.2 unwatch

取消watch命令对所有key的监视

如果在执行watch命令之后，exec命令或discard命令先执行的话，那么就不需要再执行unwatch。

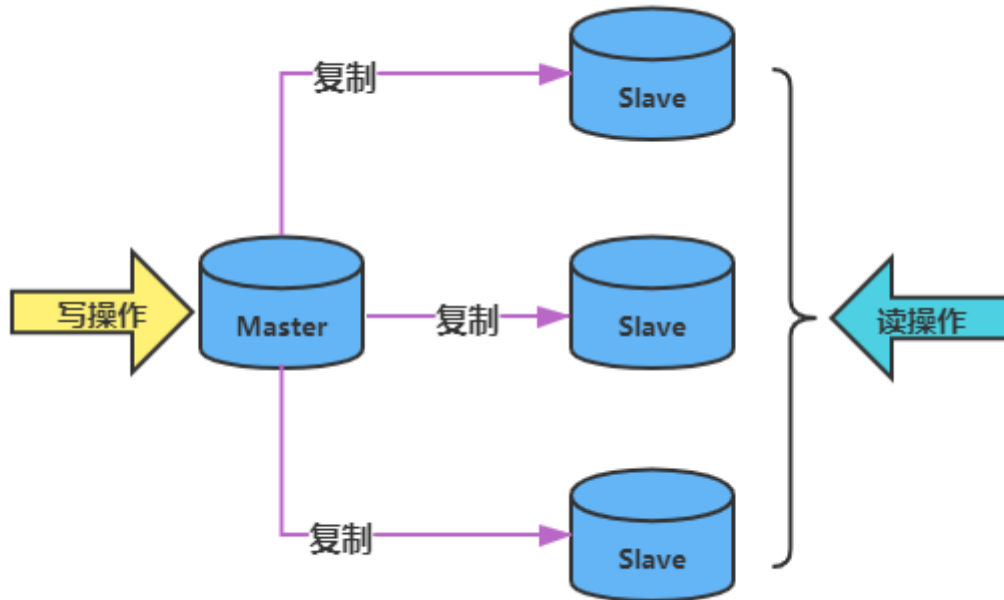
五、Redis主从复制

1 Redis主从复制简介

主从复制，是指将一台Redis服务器的数据，复制到其他的Redis服务器。前者称为主节点(Master)，后者称为从节点(Slave)；数据的复制是单向的，只能由主节点到从节点。

默认情况下，每台Redis服务器都是主节点；且一个主节点可以有多个从节点(或没有从节点)，但一个从节点只能有一个主节点。

Master以写为主，Slave以读为主。



主从复制的作用：

1. 数据冗余：主从复制实现了数据的热备份，是持久化之外的一种数据冗余方式。
2. 故障恢复：当主节点出现问题时，可以由从节点提供服务，实现快速的故障恢复；实际上是一种服务的冗余。
3. 负载均衡：在主从复制的基础上，配合读写分离，可以由主节点提供写服务，由从节点提供读服务（即写Redis数据时应用连接主节点，读Redis数据时应用连接从节点），分担服务器负载；尤其是在写少读多的场景下，通过多个从节点分担读负载，可以大大提高Redis服务器的并发量。
4. 高可用基石：除了上述作用以外，主从复制还是哨兵和集群能够实施的基础，因此说主从复制是Redis高可用的基础。

一般来说，要将Redis运用于工程项目中，不会只有一台Redis，原因如下：

1. 从结构上，单个redis服务器会发生单点故障，并且一台服务器需要处理所有的请求负载，压力较大；
2. 从容量上，单个redis服务器内存容量有限，就算一台Redis服务器内存容量为256G，也不能将所有内存用做Redis存储内存，一般来说，**单台Redis最大使用内存不应该超过20G**。

2 Redis主从复制-一主多从

搭建一主二从的Redis服务器

环境搭建

1. 在同一台虚拟机上配置一主二从Redis服务器，由于单台机器，同一个端口只允许一个进程占用，所以需要修改其他两台Redis的端口。
2. 在/usr/local/myredis文件夹下，新建三个redis的配置文件，分别为redis6379.conf、redis6380.conf、redis6381.conf，由文件名可知，redis的端口号分别为6379、6380、6381。


```
[root@localhost myredis]# ll
总用量 92
-rw-r--r--. 1 root root 0 7月 14 22:48 redis6379.conf
-rw-r--r--. 1 root root 0 7月 14 22:48 redis6380.conf
-rw-r--r--. 1 root root 0 7月 14 22:48 redis6381.conf
-rw-r--r--. 1 root root 93737 7月 6 01:32 redis.conf
[root@localhost myredis]#
```

创建出3个空文件，redis.conf是原有的配置文件。

3. 在三个配置文件写入内容

- 可以通过include /usr/local/myredis/redis.conf将公共基础配置直接引入文件。
将include /usr/local/myredis/redis.conf统一添加到这三个文件中
- 在各个文件中，添加对应的pidfile、port、dbfilename

如：

redis6379.conf中添加

```
pidfile /var/run/redis_6379.pid
port 6379
dbfilename dump6379.rdb
```

```
include /usr/local/myredis/redis.conf
pidfile /var/run/redis_6379.pid
port 6379
dbfilename dump6379.rdb
~
~
~
~
~
```

redis6380.conf中添加

```
pidfile /var/run/redis_6380.pid
port 6380
dbfilename dump6380.rdb
```

```
include /usr/local/myredis/redis.conf
pidfile /var/run/redis_6380.pid
port 6380
dbfilename dump6380.rdb
~
~
~
~
~
```

redis6381.conf中添加

```
pidfile /var/run/redis_6381.pid
port 6381
dbfilename dump6381.rdb
```

```
include /usr/local/myredis/redis.conf
pidfile /var/run/redis_6381.pid
port 6381
dbfilename dump6381.rdb
~
~
~
~
~
~
```

4. 通过不同的redis.conf文件，分别启动3个redis服务

```
./redis-server /usr/local/myredis/redis6379.conf
./redis-server /usr/local/myredis/redis6380.conf
./redis-server /usr/local/myredis/redis6381.conf
```

```
[root@localhost bin]# ./redis-server /usr/local/myredis/redis6379.conf
[root@localhost bin]# ./redis-server /usr/local/myredis/redis6380.conf
[root@localhost bin]# ./redis-server /usr/local/myredis/redis6381.conf
[root@localhost bin]#
```

通过`ps -ef | grep redis`命令查看redis服务，3个服务均已启动

```
[root@localhost bin]# ps -ef | grep redis
root      2791      1    0 23:24 ?        00:00:00 ./redis-server 127.0.0.1:6379
root      2850      1    0 23:29 ?        00:00:00 ./redis-server 127.0.0.1:6380
root      2856      1    0 23:29 ?        00:00:00 ./redis-server 127.0.0.1:6381
root      2879    2163    0 23:30 pts/0    00:00:00 grep --color=auto redis
[root@localhost bin]#
```

5. 在redis客户端中，通过info replication命令可以查看Redis服务器当前状态

可以给redis-cli命令添加-p参数，来指定链接哪个服务器

```
./redis-cli -p 6379
./redis-cli -p 6380
./redis-cli -p 6381
```

[illegible]

所有的服务器目前都是master

6. 配从 (6380、6381) 不配主 (6379)

slaveof ip port成为某个实例的从服务器。

在从机6380和6381上执行: `slaveof 127.0.0.1 6379`

在主库 (6379) 中再次执行info replication

```
127.0.0.1:6379> info replication
# Replication
role:master
connected_slaves:2
slave0:ip=127.0.0.1,port=6380,state=online,offset=210,lag=0
slave1:ip=127.0.0.1,port=6381,state=online,offset=210,lag=0
master_failover_state:no-failover
master_replid:4fa2ad14066f41496b3bb2ec4e78447fdc36736e
master_replid2:0000000000000000000000000000000000000000
master_repl_offset:210
second_repl_offset:-1
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:1
repl_backlog_histlen:210
127.0.0.1:6379> █
```

有两个从库，分别为6380和6381。

7. 在主库（6379）中写数据，可以在从库（6380、6381）中读取到。

注意：

- 在从库中进行写操作，会报错。
- 由于该主从复制在同一台虚拟机上搭建，所以需要修改端口号，如果在多台服务器上搭建主从复制，则需要修改相对应的ip。
- 如果在多台服务器上搭建主从复制，一定要开放远程链接。

3 Redis主从复制-复制原理

3.1 主从复制的一些问题

1. 如果Master断开（宕机），Slave依然连接着Master，可以正常使用读操作，但是没有写操作。如果Master恢复正常，Slave依旧可以直接获取Master写的信息。
2. 如果Slave断开（宕机），当该Slave重启成功，则会变为Master，需要通过slaveof 恢复成Slave，只要变为Slave，立刻可以从Master同步所有数据。

3.2 复制原理

- Slave启动成功连接到Master后会主动发送一个同步（sync）命令。
- Master接到Slave的命令，把Master数据进行持久化，把rdb文件发送给Slave，Slave拿到rdb进行读取。
- 每次Master进行写操作之后，会和Slave进行数据同步。



- **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。
- **增量复制**：指Slave初始化后开始正常工作时主服务器发生的写操作同步到从服务器的过程。

3.3 薪火相传

上一个Slave（从机）是下一个Slave（从机）的Master（主机）。



优点：Slave同样可以接收其他Slave的连接和同步请求，那么该Slave作为了链条中下一个的Master, 可以有效减轻Master的压力，去中心化降低风险。

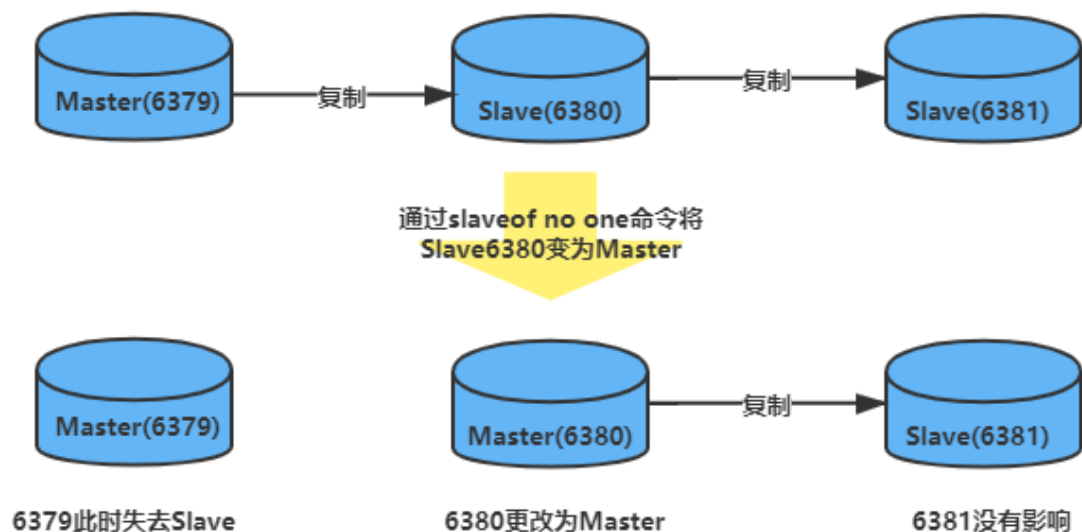
缺点：一旦某个Slave宕机，后面的Slave都无法备份。

注意：

- 也是通过`slaveof ip port`命令修改Master。
- 中途变更转向:会清除之前的数据，重新建立拷贝最新的。
- Slave6380本质上仍然是从库，只能读、不能写。

3.4 反客为主

当一个Master宕机后，后面的Slave可以立刻升为Master，其后面的Slave不用做任何修改。

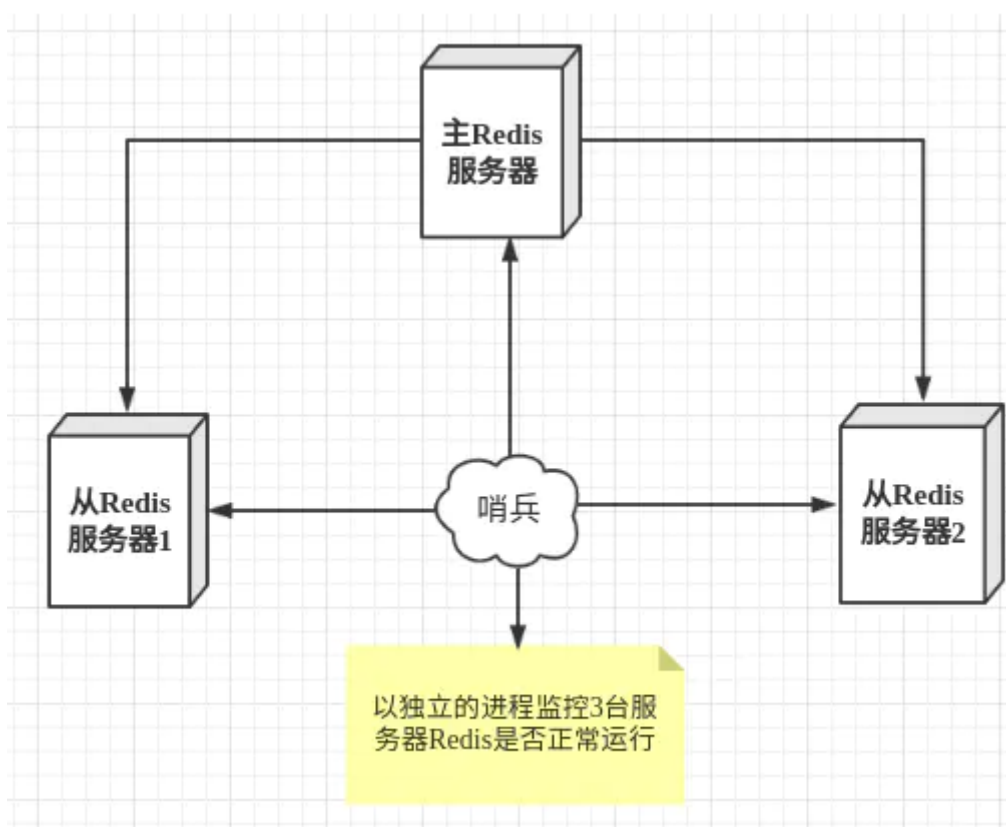


通过slaveof no one 将Slave变为Master。

4 Redis主从复制-哨兵模式 (Sentinel)

反客为主的自动版，能够后台监控Master是否故障，如果故障了，根据投票数自动将Slave转换为Master。

哨兵模式是一种特殊的模式，首先Redis提供了哨兵的命令，哨兵是一个独立的进程，作为进程，它会独立运行。其原理是哨兵通过发送命令，等待Redis服务器响应，从而监控运行的多个Redis实例。



4.1 哨兵模式的使用

1. 将服务器调整为一主多从（6379带6380、6381）。

```
127.0.0.1:6379> info replication
# Replication
role:master
connected_slaves:2
slave0:ip=127.0.0.1,port=6380,state=online,offset=140,lag=1
slave1:ip=127.0.0.1,port=6381,state=online,offset=140,lag=1
master_failover_state:no-failover
master_replid:27a9fe5424cecd7a8ccbec700d27519c93e12e37
master_replid2:0000000000000000000000000000000000000000
master_repl_offset:140
second_repl_offset:-1
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:1
repl_backlog_histlen:140
```

2. 在/usr/local/myredis文件夹下创建sentinel.conf文件（名字一定不能错）。

```
[root@localhost myredis]# ls
redis6379.conf redis6380.conf redis6381.conf redis.conf sentinel.conf
[root@localhost myredis]#
```

3. 配置哨兵，sentinel.conf添加内容

```
# sentinel monitor <master-name> <ip> <port> <count>w
sentinel monitor myredis 127.0.0.1 6379 1
```

其中：myredis 为监控对象起的服务器名称（随意），1代表至少有1个哨兵投票同意迁移

如果Master存在密码，需要配置sentinel auth-pass 服务器名 密码

```
sentinel auth-pass myredis 123456
daemonize yes
```

```
sentinel monitor myredis 127.0.0.1 6379 1
~
~
~
```

4. 启动哨兵，在/usr/local/bin/文件夹下执行./redis-sentinel /usr/local/myredis/sentinel.conf命令。

```
[root@localhost bin]# ./redis-sentinel /usr/local/myredis/sentinel.conf
5948:X 15 Jul 2021 05:07:47.328 # oO00oO00oO00o Redis is starting oO00oO00oO00o
5948:X 15 Jul 2021 05:07:47.328 # Redis version=6.2.4, bits=64, commit=00000000, modified=0, pid=5948, just started
5948:X 15 Jul 2021 05:07:47.328 # Configuration loaded
5948:X 15 Jul 2021 05:07:47.328 * Increased maximum number of open files to 10032 (it was originally set to 1024).
5948:X 15 Jul 2021 05:07:47.328 * monotonic clock: POSIX clock_gettime

Redis 6.2.4 (00000000/0) 64 bit
Running in sentinel mode
Port: 26379
PID: 5948

https://redis.io

5948:X 15 Jul 2021 05:07:47.328 # WARNING: The TCP backlog setting of 511 cannot be enforced because /proc/sys/net/core/somaxconn is set to the lower value of 128.
5948:X 15 Jul 2021 05:07:47.329 # Sentinel ID is fa0e39fab561ab0a81fe871462e0e21e8a61fa88
5948:X 15 Jul 2021 05:07:47.329 # +monitor master myredis 127.0.0.1 6379 quorum 1
```

5. 模拟Master宕机，哨兵会切换Master

- 通过kill命令关闭Master（6379）

```
[root@localhost bin]# ps -ef | grep redis
root      2850      1   0  7月14 ?        00:00:10 ./redis-server 127.0.0.1:6380
root      2856      1   0  7月14 ?        00:00:10 ./redis-server 127.0.0.1:6381
root      5451      1   0  04:18 ?        00:00:01 ./redis-server 127.0.0.1:6379
root      5988    2163   0  05:11 pts/0    00:00:00 ./redis-sentinel *:26379 [sentinel]
root      5994    4986   0  05:12 pts/1    00:00:00 grep --color=auto redis
[root@localhost bin]# kill -9 5451
[root@localhost bin]#
```

- 等待一段时间，哨兵窗口就会输出信息

```
[root@localhost bin]# ./redis-sentinel /usr/local/myredis/sentinel.conf
5988:X 15 Jul 2021 05:11:54.077 # 000000000000 Redis is starting 000000000000
5988:X 15 Jul 2021 05:11:54.077 # Redis version=6.2.4, bits=64, commit=00000000, modified=0, pid=5988, just started
5988:X 15 Jul 2021 05:11:54.077 # Configuration loaded
5988:X 15 Jul 2021 05:11:54.077 * Increased maximum number of open files to 10032 (it was originally set to 1024).
5988:X 15 Jul 2021 05:11:54.077 * monotonic clock: POSIX clock_gettime

Redis 6.2.4 (00000000/0) 64 bit

Running in sentinel mode
Port: 26379
PID: 5988

https://redis.io

5988:X 15 Jul 2021 05:11:54.078 # WARNING: The TCP backlog setting of 511 cannot be enforced because /proc/sys/net/core/somaxconn is set to the lower value of 128.
5988:X 15 Jul 2021 05:11:54.078 # Sentinel ID is fa0e39fab561ab0a81fe071462e0e21e8a61fa88
5988:X 15 Jul 2021 05:11:54.078 # Monitor master myredis 127.0.0.1 6379 quorum 1
5988:X 15 Jul 2021 05:13:22.620 # +sdown master myredis 127.0.0.1 6379
5988:X 15 Jul 2021 05:13:22.620 # +down master myredis 127.0.0.1 6379 #quorum 1/1
5988:X 15 Jul 2021 05:13:22.620 # +new-epoch 1
5988:X 15 Jul 2021 05:13:22.620 # +try-failover master myredis 127.0.0.1 6379
5988:X 15 Jul 2021 05:13:22.621 # +vote-for-leader fa0e39fab561ab0a81fe071462e0e21e8a61fa88 1
5988:X 15 Jul 2021 05:13:22.621 # +selected-leader master myredis 127.0.0.1 6379
5988:X 15 Jul 2021 05:13:22.621 # +failover-state-select-slave master myredis 127.0.0.1 6379
5988:X 15 Jul 2021 05:13:22.677 # +selected-slave slave 127.0.0.1:6381 127.0.0.1 6381 @ myredis 127.0.0.1 6379
5988:X 15 Jul 2021 05:13:22.738 # +failover-state-send-slaveof-noone slave 127.0.0.1:6381 127.0.0.1 6381 @ myredis 127.0.0.1 6379
5988:X 15 Jul 2021 05:13:23.463 # +promoted-slave slave 127.0.0.1:6381 127.0.0.1 6381 @ myredis 127.0.0.1 6379
5988:X 15 Jul 2021 05:13:23.463 # +failover-state-reconf-slaves master myredis 127.0.0.1 6379
5988:X 15 Jul 2021 05:13:23.553 # +slave-reconf-sent slave 127.0.0.1:6380 127.0.0.1 6380 @ myredis 127.0.0.1 6379
5988:X 15 Jul 2021 05:13:24.480 # +slave-reconf-inprog slave 127.0.0.1:6380 127.0.0.1 6380 @ myredis 127.0.0.1 6379
5988:X 15 Jul 2021 05:13:24.480 # +slave-reconf-done slave 127.0.0.1:6380 127.0.0.1 6380 @ myredis 127.0.0.1 6379
5988:X 15 Jul 2021 05:13:24.547 # +failover-end master myredis 127.0.0.1 6379
5988:X 15 Jul 2021 05:13:24.547 # +switch-master myredis 127.0.0.1 6379 127.0.0.1 6381
5988:X 15 Jul 2021 05:13:24.547 # +slave slave 127.0.0.1:6380 127.0.0.1 6380 @ myredis 127.0.0.1 6381
5988:X 15 Jul 2021 05:13:24.547 # +slave slave 127.0.0.1:6379 127.0.0.1 6379 @ myredis 127.0.0.1 6381
5988:X 15 Jul 2021 05:13:54.627 # +sdown slave 127.0.0.1:6379 127.0.0.1 6379 @ myredis 127.0.0.1 6381
```

根据窗口信息可知，Master由6379转换为6381，进入6381客户端执行info replication

```
[root@localhost bin]# ./redis-cli -p 6381
127.0.0.1:6381> info replication
# Replication
role:master
connected_slaves:1
slave0:ip=127.0.0.1,port=6380,state=online,offset=49078,lag=1
master_failover_state:no-failover
master_replid:d9193b4595aeeb7408db8c8a8682ed97a31543a9
master_replid2:27a9fe5424cecd7a8ccbec700d27519c93e12e37
master_repl_offset:49210
second_repl_offset:23576
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:127
repl_backlog_histlen:49084
127.0.0.1:6381>
```

6. 重新启动6379服务器后，自动转换为Slave。

```
[root@localhost bin]# ./redis-server ../myredis/redis6379.conf
[root@localhost bin]# ./redis-cli -p 6379
127.0.0.1:6379> info replication
# Replication
role:slave
master host:127.0.0.1
master port:6381
master_link_status:up
master_last_io_seconds_ago:1
master_sync_in_progress:0
slave_repl_offset:59320
slave_priority:100
slave_read_only:1
replica_announced:1
connected_slaves:0
master_failover_state:no-failover
master_replid:d9193b4595aeeb7408db8c8a8682ed97a31543a9
master_replid2:0000000000000000000000000000000000000000
master_repl_offset:59320
second_repl_offset:-1
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:59321
repl_backlog_histlen:0
127.0.0.1:6379>
```

4.2 复制延迟

由于所有的写操作都是先在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，Slave机器数量的增加，会使延迟问题会更加严重。

六、Redis集群

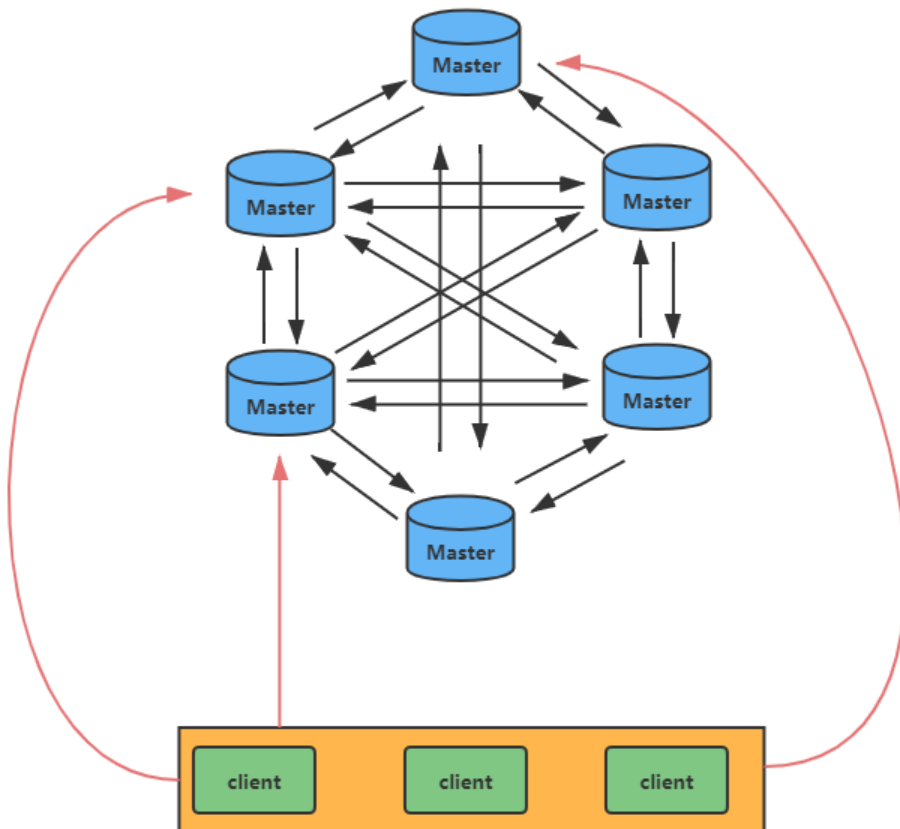
1 Redis集群简介

1.1 Redis集群 (RedisCluster)

RedisCluster实现了对Redis的水平扩容，即启动N个Redis节点，将整个数据库分布存储在这N个节点当中，每个节点存储总数据的1/N。

RedisCluster通过分区（partition）来提供一定程度的可用性（availability）：即集群有一部分节点失效或者无法进行通讯，集群也可以继续处理命令。

Redis-Cluster采用无中心结构，每个节点保存数据和整个集群状态，每个节点都和其他所有节点连接，用来交换彼此的信息。



为了使得集群在一部分节点宕机或者无法与集群的大多数节点进行通讯的情况下，仍然可以正常运作，Redis 集群对节点使用了主从复制功能。

2 Redis集群搭建

1. 删除持久化数据，aof、rdb。
2. 创建6个实例，即在/usr/local/myredis文件夹下分别创建6379、6380、6381、6389、6390、6391的conf。
3. 在redis.conf中关闭appendonly。
4. 在不同的conf文件中，配置对应内容
 - 可以通过include /usr/local/myredis/redis.conf将公共基础配置直接引入文件。
将include /usr/local/myredis/redis.conf统一添加到这三个文件中
 - 在各个文件中，添加对应的pidfile、port、dbfilename、cluster-enabled（是否打开集群）、cluster-config-file（设定节点配置文件名）、cluster-node-timeout（设置节点失联时间，超过该时间（毫秒），集群自动进行主从切换）

如：

redis6379.conf中添加

```
pidfile /var/run/redis_6379.pid
port 6379
dbfilename dump6379.rdb
# 是否打开集群
cluster-enabled yes
# 设定节点配置文件名
cluster-config-file nodes-6379.conf
# 设置节点失联时间，超过该时间（毫秒），集群自动进行主从切换
cluster-node-timeout 15000
```

```
include /usr/local/myredis/redis.conf
pidfile "/var/run/redis_6379.pid"
port 6379
dbfilename "dump6379.rdb"
cluster-enabled yes
cluster-config-file nodes-6379.conf
cluster-node-timeout 15000
```

将其他文件按照相同规则创建出来。

5. 启动该6个redis，并确保是否全部生成nodes-xxxx.conf文件。（启动之前务必保证redis服务支持远程连接）

```
[root@localhost bin]# ./redis-server ../myredis/redis6379.conf
[root@localhost bin]# ./redis-server ../myredis/redis6380.conf
[root@localhost bin]# ./redis-server ../myredis/redis6381.conf
[root@localhost bin]# ./redis-server ../myredis/redis6389.conf
[root@localhost bin]# ./redis-server ../myredis/redis6390.conf
[root@localhost bin]# ./redis-server ../myredis/redis6391.conf
[root@localhost bin]# ps -ef | grep redis
root      15163      1  0 22:47 ?        00:00:00 ./redis-server 127.0.0.1:6379 [cluster]
root      15169      1  0 22:47 ?        00:00:00 ./redis-server 127.0.0.1:6380 [cluster]
root      15175      1  0 22:47 ?        00:00:00 ./redis-server 127.0.0.1:6381 [cluster]
root      15181      1  0 22:47 ?        00:00:00 ./redis-server 127.0.0.1:6389 [cluster]
root      15187      1  0 22:47 ?        00:00:00 ./redis-server 127.0.0.1:6390 [cluster]
root      15201      1  0 22:47 ?        00:00:00 ./redis-server 127.0.0.1:6391 [cluster]
root      15207      0  0 22:48 pts/1    00:00:00 grep --color=auto redis
```

```
[root@localhost bin]# ll
总用量 24864
-rw-r--r--. 1 root root      114 7月  15 22:47 nodes-6379.conf
-rw-r--r--. 1 root root      114 7月  15 22:47 nodes-6380.conf
-rw-r--r--. 1 root root      114 7月  15 22:47 nodes-6381.conf
-rw-r--r--. 1 root root      114 7月  15 22:40 nodes-6389.conf
-rw-r--r--. 1 root root      114 7月  15 22:40 nodes-6390.conf
-rw-r--r--. 1 root root      114 7月  15 22:40 nodes-6391.conf
-rwxr-xr-x. 1 root root 6549080 6月  28 04:55 redis-benchmark
lrwxrwxrwx. 1 root root      12 6月  28 04:55 redis-check-aof -> redis-server
lrwxrwxrwx. 1 root root      12 6月  28 04:55 redis-check-rdb -> redis-server
-rwxr-xr-x. 1 root root 6763376 6月  28 04:55 redis-cli
lrwxrwxrwx. 1 root root      12 6月  28 04:55 redis-sentinel -> redis-server
-rwxr-xr-x. 1 root root 12118288 6月  28 04:55 redis-server
```

6. 进入redis安装目录下的src文件夹

```
[root@localhost src]# pwd
/download/redis-6.2.4/src
[root@localhost src]#
```

7. 在该文件夹下执行命令

```
redis-cli --cluster create --cluster-replicas 1 192.168.27.130:6379
192.168.27.130:6380
192.168.27.130:6381 192.168.27.130:6389 192.168.27.130:6390
192.168.27.130:6391
```

此处使用真实ip地址，-replicas 1代表采用最简单的方式配置集群，一台主机，一台从机。

replicas表示每个master需要有几个slave。

```
[root@localhost src]# redis-cli --cluster create --cluster-replicas 1 192.168.56.31:6379 192.168.56.31:6380
>>> Performing hash slots allocation on 6 nodes...
Master[0] -> Slots 0 - 5460
Master[1] -> Slots 5461 - 10922
Master[2] -> Slots 10923 - 16383
Adding replica 192.168.56.31:6390 to 192.168.56.31:6379
Adding replica 192.168.56.31:6391 to 192.168.56.31:6380
Adding replica 192.168.56.31:6389 to 192.168.56.31:6381
>>> Trying to optimize slaves allocation for anti-affinity
[WARNING] Some slaves are in the same host as their master
M: 986a9e0c374bfe0d3d826b9b33c3dde0e12133a 192.168.56.31:6379
slots:[0-5460] (5461 slots) master
M: 2c7a3ddd9b921862119600d982615f417b4a7a0b 192.168.56.31:6380
slots:[5461-10922] (5462 slots) master
M: 8e8aa41f0dd2b66c8a2348c377b0a751e8d53c13 192.168.56.31:6381
slots:[10923-16383] (5461 slots) master
S: b2022abc7cf5b6ed50f8f5297bfdab2cb6e24f9e 192.168.56.31:6389
replicates 8e8aa41f0dd2b66c8a2348c377b0a751e8d53c13
S: ffc82180b0ddef727b83cb193995d515188f47c11 192.168.56.31:6390
replicates 986a9e0c374bfe0d3d826b9b33c3dde0e12133a
S: 52033b4a26de0ad39a82889a1594e5804324abe1 192.168.56.31:6391
replicates 2c7a3ddd9b921862119600d982615f417b4a7a0b
Can I set the above configuration? (type 'yes' to accept): yes
```

执行命令后，redis提供推荐的主从配置建议，执行同意。

```
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.
```

集群创建成功。

- 通过./redis-cli -c -p 6379可连接至集群（由于所有节点相通，任意端口号均可）

在redis客户端中执行cluster nodes查看节点状态。

```
[root@localhost bin]# ./redis-cli -c -p 6379
127.0.0.1:6379> cluster nodes
8e8aa41f0dd2b66c8a2348c377b0a751e8d53c13 192.168.56.31:6381@16381 master - 0 1626405106000 3 connected 10923-16383
2c7a3ddd9b921862119600d982615f417b4a7a0b 192.168.56.31:6380@16380 master - 0 1626405106961 2 connected 5461-10922
52033b4a26de0ad39a82889a1594e5804324abe1 192.168.56.31:6391@16391 slave 2c7a3ddd9b921862119600d982615f417b4a7a0b 0 1626405106000 2 connected
986a9e0c374bfe0d3d826b9b33c3dde0e12133a 192.168.56.31:6379@16379 myself,master - 0 1626405105000 1 connected 0-5460
ffc82180b0ddef727b83cb193995d515188f47c11 192.168.56.31:6390@16390 slave 986a9e0c374bfe0d3d826b9b33c3dde0e12133a 0 1626405104000 1 connected
b2022abc7cf5b6ed50f8f5297bfdab2cb6e24f9e 192.168.56.31:6389@16389 slave 8e8aa41f0dd2b66c8a2348c377b0a751e8d53c13 0 1626405105954 3 connected
127.0.0.1:6379>
```

3 Redis集群操作

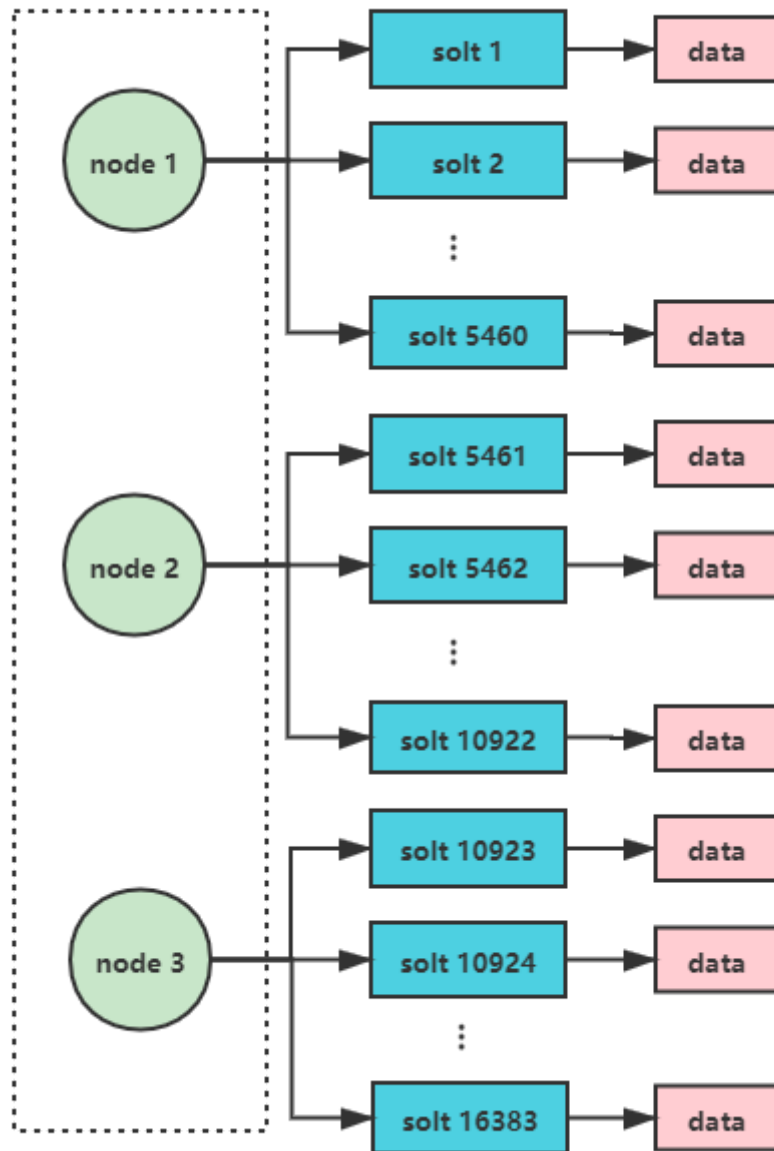
3.1 Slot

一个 Redis 集群包含 16384 个哈希槽（hash slot），每个键都属于这 16384 个哈希槽的其中一个。

集群使用公式 $CRC16(key) \% 16384$ 来计算键 key 属于哪个槽，其中 $CRC16(key)$ 语句用于计算键 key 的 CRC16 校验和。

集群中的每个节点负责处理一部分哈希槽。举个例子，一个集群可以有三个节点，其中：

- 节点 1 负责处理 0 号至 5500 号哈希槽。
- 节点 2 负责处理 5501 号至 11000 号哈希槽。
- 节点 3 负责处理 11001 号至 16384 号哈希槽。



3.2 命令执行

```
set k1 v1
```

```
127.0.0.1:6379> set k1 v1  
-> Redirected to slot [12706] located at 192.168.56.31:6381  
OK  
192.168.56.31:6381> █
```

根据k1计算出的槽值进行切换节点，并存入数据。

不在一个slot下的键值，是不能使用mget、mset等多键操作。

```
192.168.56.31:6381> mset k1 v1 k2 v2 k3 v3  
(error) CROSSSLOT Keys in request don't hash to the same slot  
192.168.56.31:6381> █
```

可以通过{}来定义组的概念，从而是key中{}内相同内容的键值对放到同一个slot中

```
mset k1{test} v1 k2{test} v2 k3{test} v3
```

```
192.168.56.31:6381> mset k1{test} v1 k2{test} v2 k3{test} v3
-> Redirected to slot [6918] located at 192.168.56.31:6380
OK
192.168.56.31:6380>
```

3.3 故障恢复

1. 关闭6379服务器，进入redis-cli执行cluster nodes

```
127.0.0.1:6380> cluster nodes
ffcf82180bdf727b83cb193995d515188f47c11 192.168.56.31:6390@16390 master - 0 1626407518900 7 connected 0-5460
b2022abc7cf5b6ed50f8f5297bfdab2cb6e24f9e 192.168.56.31:6389@16389 slave 8e8aa41f0dd2b66c8a2348c377b0a751e8d53c13 0 1626407516000 3 connect
ed
52033b4a26de0ad39a82889a1594e5804324abe1 192.168.56.31:6391@16391 slave 2c7a3ddd9b921862119600d982615f417b4a7a0b 0 1626407518875 2 connect
ed
8e8aa41f0dd2b66c8a2348c377b0a751e8d53c13 192.168.56.31:6381@16381 master - 0 1626407517866 3 connected 10923-16383
2c7a3ddd9b921862119600d982615f417b4a7a0b 192.168.56.31:6380@16380 myself,master - 0 1626407516000 2 connected 5461-10922
986a9e0c374bfe0d3d826b9b33c3dde0e12133a 192.168.56.31:6379@16379 master fail - 1626407473591 1626407466000 1 disconnected
```

6379服务器fail，6390成为了新的master

2. 重新启动6379服务器，再次查看，6379成为了Slave

```
127.0.0.1:6380> cluster nodes
ffcf82180bdf727b83cb193995d515188f47c11 192.168.56.31:6390@16390 master - 0 1626407647988 7 connected 0-5460
b2022abc7cf5b6ed50f8f5297bfdab2cb6e24f9e 192.168.56.31:6389@16389 slave 8e8aa41f0dd2b66c8a2348c377b0a751e8d53c13 0 1626407649000 3 connect
ed
52033b4a26de0ad39a82889a1594e5804324abe1 192.168.56.31:6391@16391 slave 2c7a3ddd9b921862119600d982615f417b4a7a0b 0 1626407648000 2 connect
ed
8e8aa41f0dd2b66c8a2348c377b0a751e8d53c13 192.168.56.31:6381@16381 master - 0 1626407648995 3 connected 10923-16383
2c7a3ddd9b921862119600d982615f417b4a7a0b 192.168.56.31:6380@16380 myself,master - 0 1626407648000 2 connected 5461-10922
986a9e0c374bfe0d3d826b9b33c3dde0e12133a 192.168.56.31:6379@16379 slave ffcf82180bdf727b83cb193995d515188f47c11 0 1626407650002 7 connect
ed
```

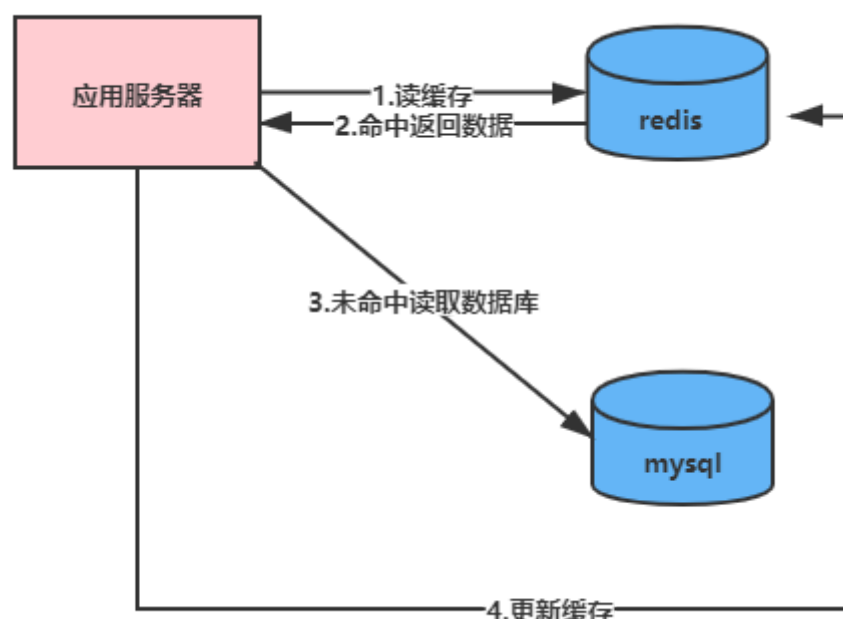
七、Python与Redis

1 redis操作Redis

1.1 Cache Aside Pattern（缓存模式）

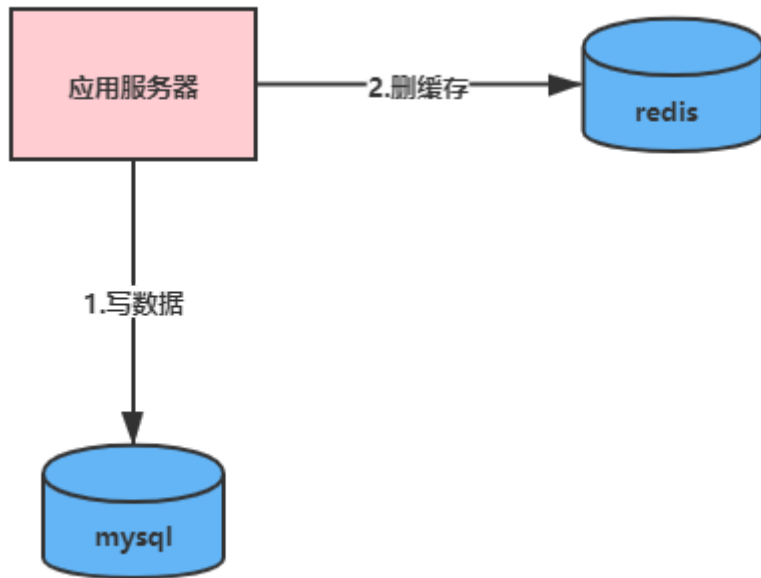
读：

1. 先读缓存，再读数据库
2. 如果缓存命中，则直接返回缓存数据
3. 如果缓存未命中，则访问数据库，并将数据重置回缓存，然后返回。



写：

先操作数据库写，再淘汰缓存（这里淘汰缓存是删除，而不是更新）



redis是Redis官方推荐的Java连接开发工具。要在Java开发中使用好Redis中间件，必须对Jedis熟悉才能写成漂亮的代码。

1.2 redis的使用

Github: <https://github.com/andymccurdy/redis-py>

安装:

```
pip install redis
```

1.3 常用方法

使用方式与redis的方法基本一致

string

- set
- setex
- mset
- append
- get
- mget
- key

keys

- exists
- type
- delete
- expire
- getrange
- ttl

list

- lpush
- rpush

- linsert
- lrange
- lset
- lrem

set

- sadd
- smembers
- srem

zset

- zadd
- zrange
- zrangebyscore
- zscore
- zrem
- zremrangebyscore

hash

- hset
- hmset
- hkeys
- hget
- hmget
- hvals
- hdel

```
import redis

r = redis.Redis()
print(r.keys())
r.set('k1', 'v1')
v1 = r.get('k1')
print(v1)

r.lpush('lkey1', '123')
print(r.lrange('lkey1', 0, -1))

r.delete('k1')
```

2 RedisCluster操作Redis集群

Github <https://github.com/Grokzen/redis-py-cluster>

安装

```
pip install redis-py-cluster
```

启动redis集群

```
[root@localhost bin]# ./redis-server ../myredis/redis6379.conf
[root@localhost bin]# ./redis-server ../myredis/redis6380.conf
[root@localhost bin]# ./redis-server ../myredis/redis6381.conf
[root@localhost bin]# ./redis-server ../myredis/redis6389.conf
[root@localhost bin]# ./redis-server ../myredis/redis6390.conf
[root@localhost bin]# ./redis-server ../myredis/redis6391.conf
```

进入redis安装目录下的src文件夹执行

```
redis-cli --cluster create --cluster-replicas 1 192.168.27.130:6379
192.168.27.130:6380
192.168.27.130:6381 192.168.27.130:6389 192.168.27.130:6390 192.168.27.130:6391
```

```
from rediscluster import RedisCluster
if __name__ == '__main__':
    try:
        # 构建所有的节点，Redis会使用CRC16算法，将键和值写到某个节点上
        startup_nodes = [
            {'host': '192.168.27.130', 'port': 6379},
            {'host': '192.168.27.130', 'port': 6380},
            {'host': '192.168.27.130', 'port': 6381},
            {'host': '192.168.27.130', 'port': 6389},
            {'host': '192.168.27.130', 'port': 6390},
            {'host': '192.168.27.130', 'port': 6391},
        ]
        # 构建StrictRedisCluster对象


        src=RedisCluster(startup_nodes=startup_nodes,decode_responses=True)
        # 设置键为name、值为sxt的数据
        result=src.set('name','sxt')
        print(result)
        # 获取键为name
        name = src.get('name')
        print(name)
    except Exception as e:
        print(e)
```

八、RedisDesktopManager

一款好用的Redis桌面管理工具，支持命令控制台操作，以及常用，查询key，rename，delete等操作。

RedisDesktopManager不支持集群操作。

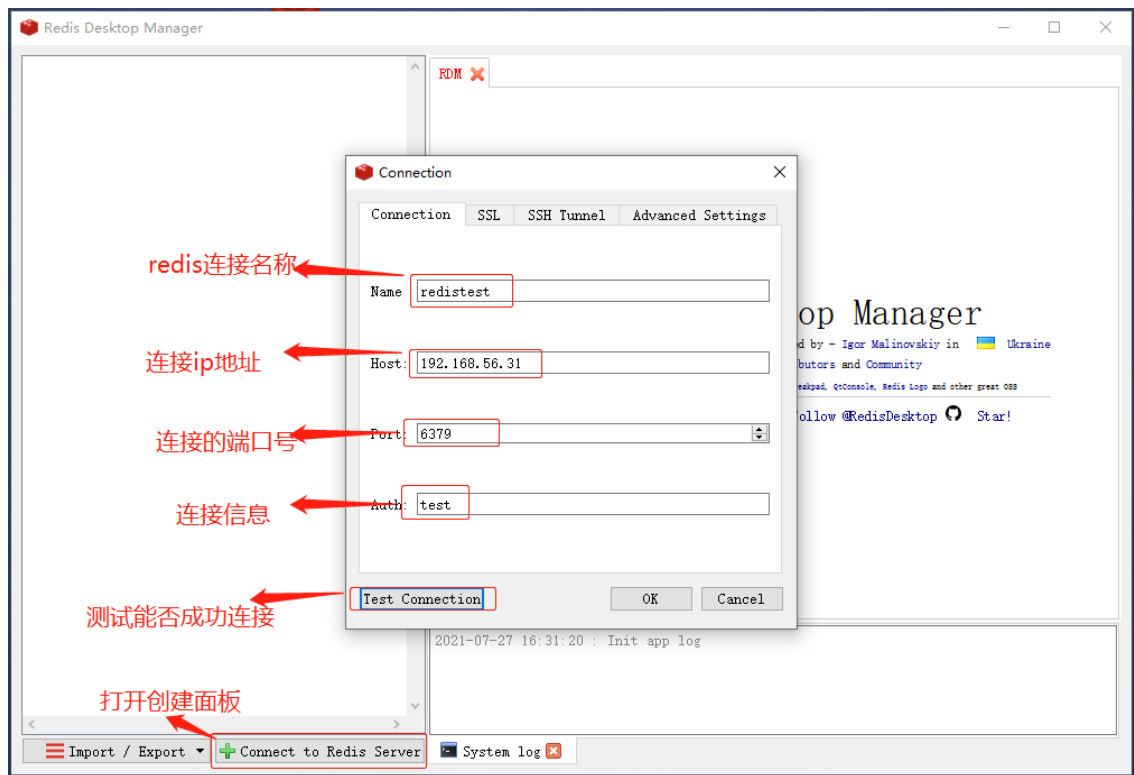
1. 傻瓜式安装该工具

 redis-desktop-manager-0.7.9.809.exe 2020/1/17 14:55

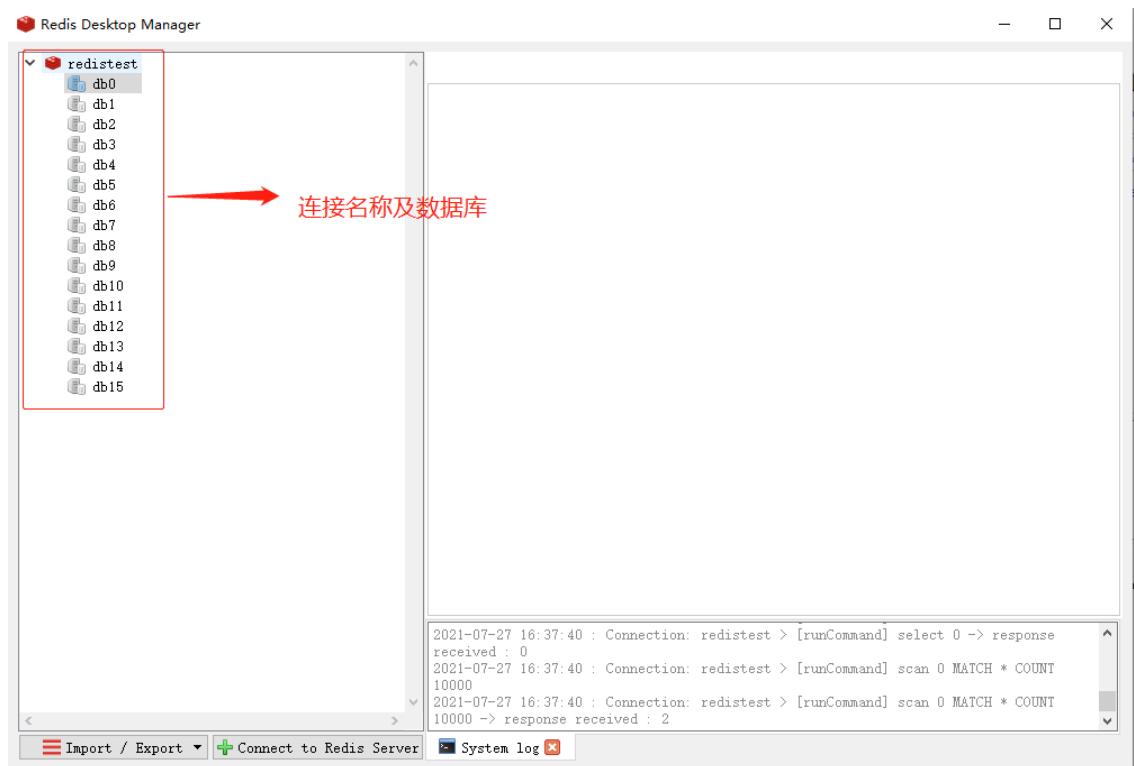
2. 安装成功后，启动该工具



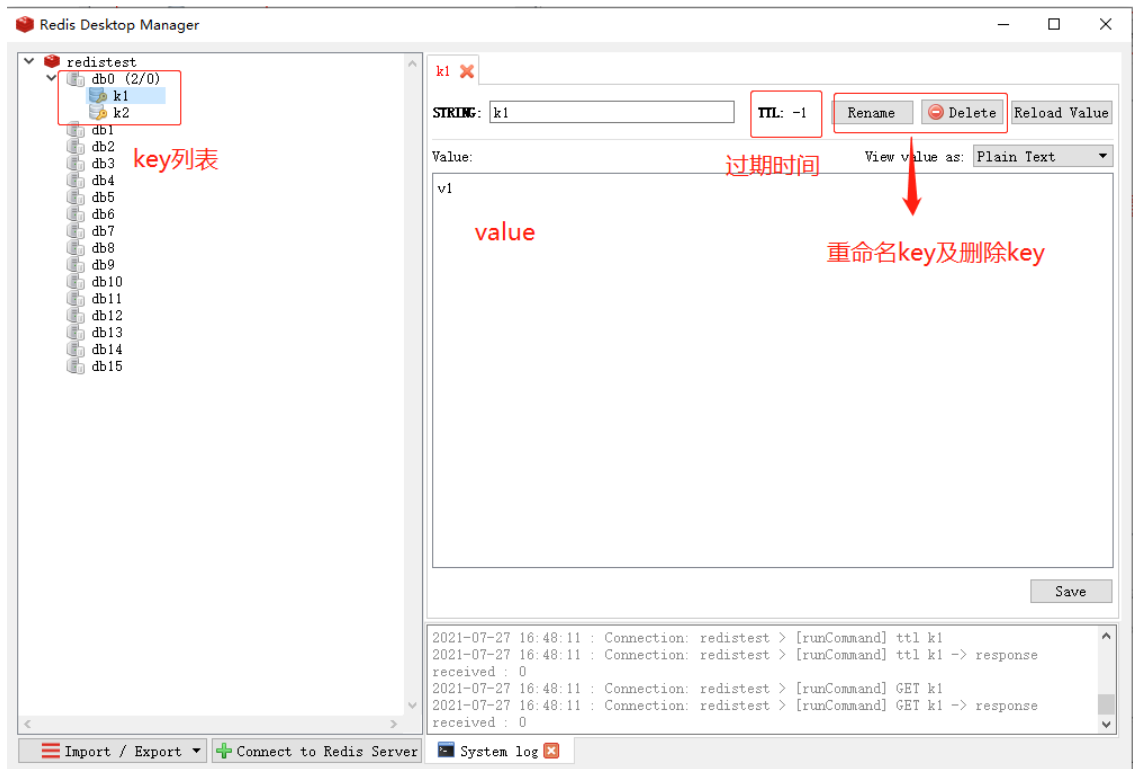
3. 启动后，创建新的连接



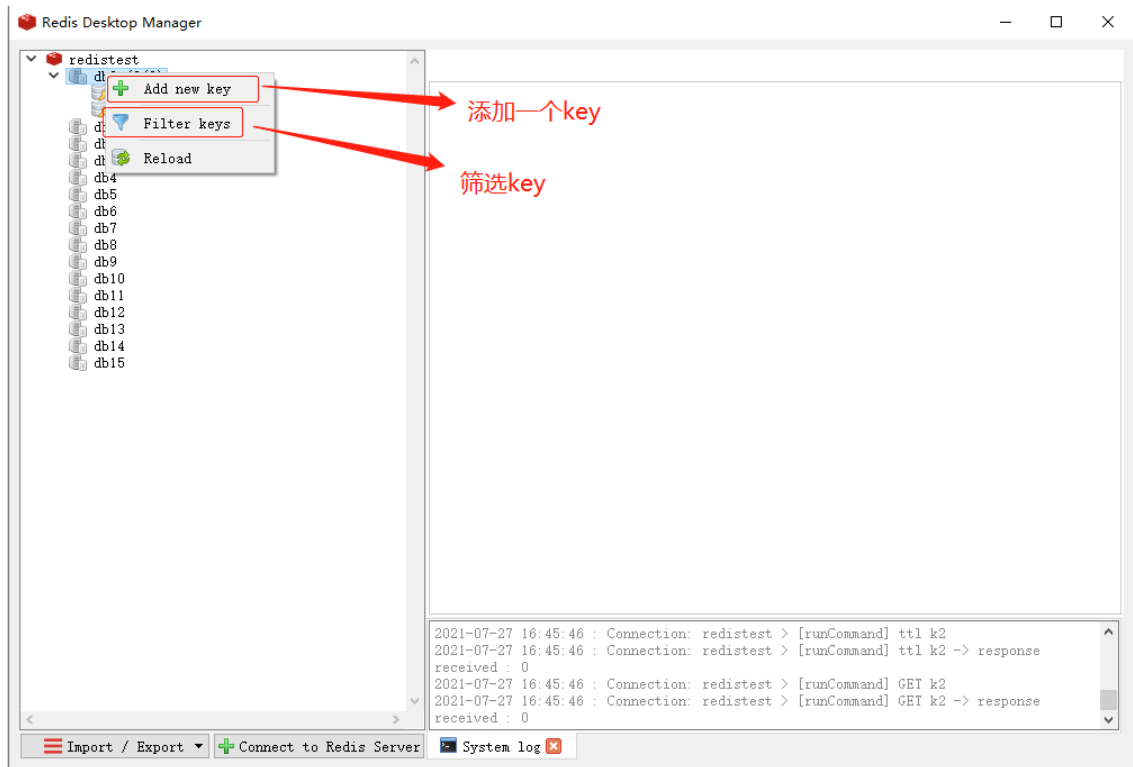
4. 创建连接成功



5. 双击进入数据库，可以查看所有的key及value、过期时间，同时可以修改key或删除key



6. 在数据库上右键可以选择新增key或筛选key



Add New Key ? X

Key: → **key**

Type: **hash**

Value:

Key:

→ **键值对 name:百战**

新增一个hash

Save Cancel



注：对key更改后(新增、修改、删除)，需要通过右键**连接名**->reload重新加载

Redis Desktop Manager

