

# 框架高级课程系列之 Redis6

尚硅谷 JavaEE 教研组

## 1. NoSQL 数据库简介

### 1.1. 技术发展

技术的分类

- 1、解决功能性的  
问题：Java、Jsp、RDBMS、Tomcat、HTML、Linux、JDBC、SVN
- 2、解决扩展性的问题：  
Struts、Spring、SpringMVC、Hibernate、Mybatis
- 3、解决性能的问题：  
NoSQL、Java 线程、Hadoop、Nginx、MQ、ElasticSearch

#### 1.1.1. Web1.0 时代

Web1.0 的时代，数据访问量很有限，用一夫当关的高性能的单点服务器可以解决大部分问题。



#### 1.1.2. Web2.0 时代

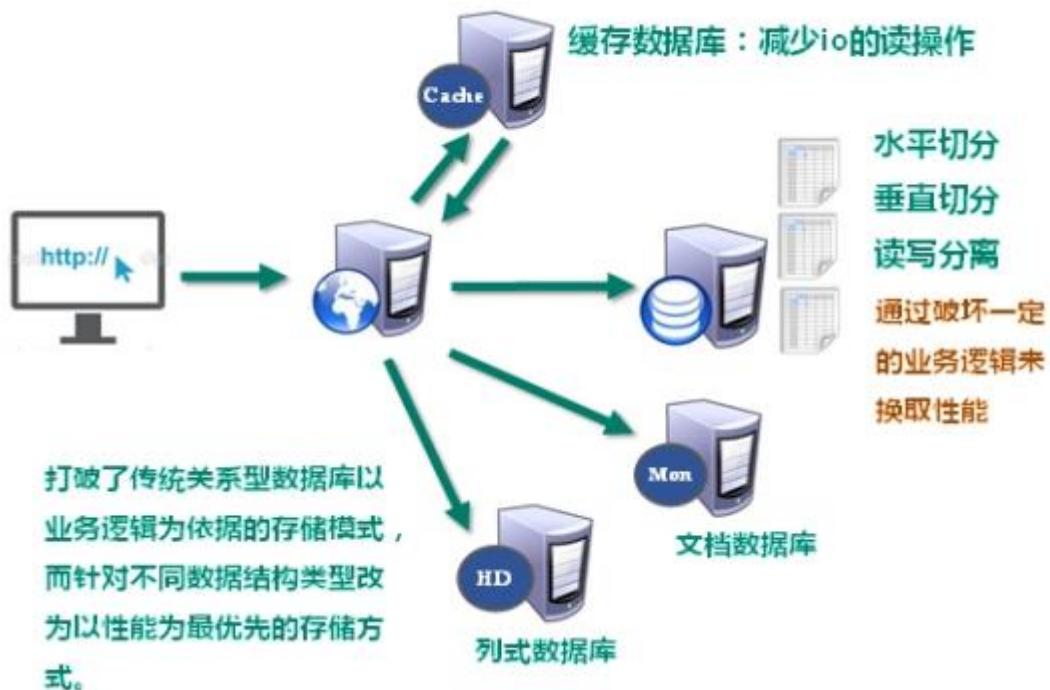
随着 Web2.0 的时代的到来，用户访问量大幅度提升，同时产生了大量的用户数据。加上后来的智能移动设备的普及，所有的互联网平台都面临了巨大的性能挑战。



### 1.1.3. 解决 CPU 及内存压力



### 1.1.4. 解决 I/O 压力



## 1.2. NoSQL 数据库

### 1.2.1. NoSQL 数据库概述

NoSQL(NoSQL = **Not Only SQL** ), 意即“不仅仅是 SQL”，泛指**非关系型的数据库**。

NoSQL 不依赖业务逻辑方式存储，而以简单的 **key-value** 模式存储。因此大大的增加了数据库的扩展能力。

- 不遵循 SQL 标准。
- 不支持 ACID。
- 远超于 SQL 的性能。

### 1.2.2. NoSQL 适用场景

- 对数据高并发的读写
- 海量数据的读写
- 对数据高可扩展性的

### 1.2.3. NoSQL 不适用场景

- 需要事务支持
- 基于 sql 的结构化查询存储，处理复杂的关系，需要即席查询。
- (用不着 sql 的和用了 sql 也不行的情况，请考虑用 NoSql)

### 1.2.4. Memcache



- ✓ 很早出现的 NoSql 数据库
- ✓ 数据都在内存中，一般不持久化
- ✓ 支持简单的 key-value 模式，支持类型单一
- ✓ 一般是作为缓存数据库辅助持久化的数据库

### 1.2.5. Redis



- ✓ 几乎覆盖了 Memcached 的绝大部分功能
- ✓ 数据都在内存中，支持持久化，主要用作备份恢复
- ✓ 除了支持简单的 key-value 模式，还支持多种数据结构的存储，比如 list、set、hash、zset 等。
- ✓ 一般是作为缓存数据库辅助持久化的数据库

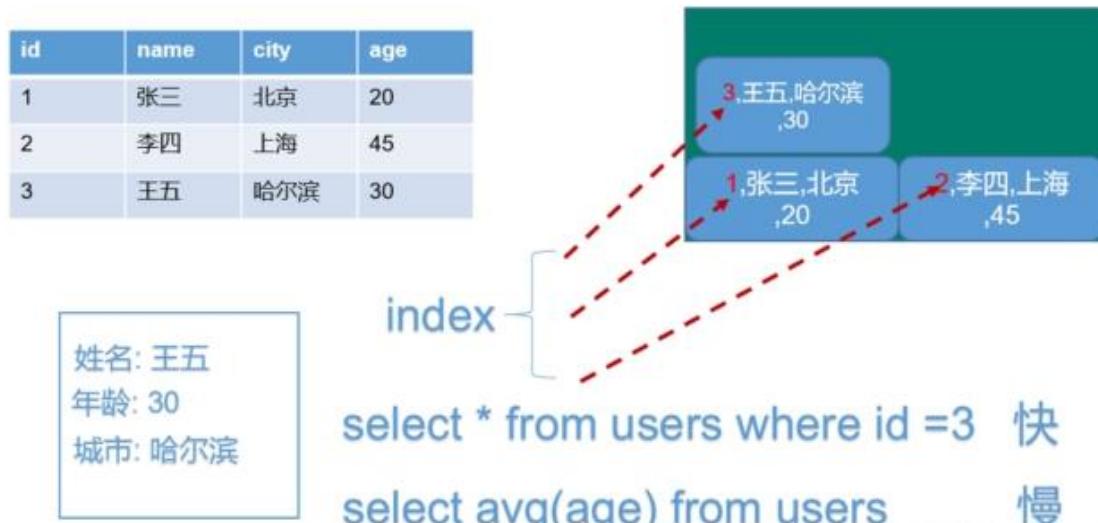
### 1.2.6. MongoDB



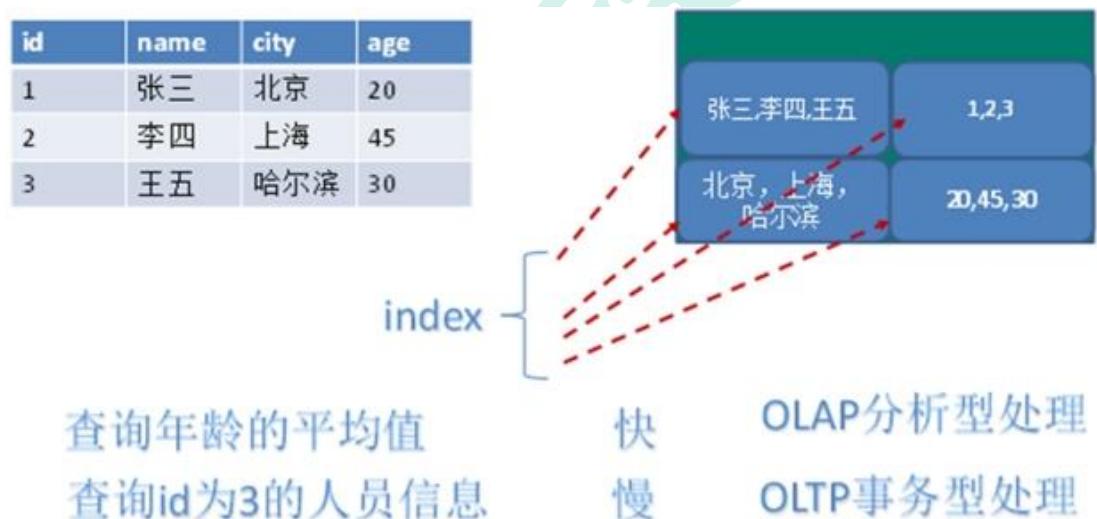
- ✓ 高性能、开源、模式自由 (schema free) 的文档型数据库
- ✓ 数据都在内存中，如果内存不足，把不常用的数据保存到硬盘
- ✓ 虽然是 key-value 模式，但是对 value (尤其是 json) 提供了丰富的查询功能
- ✓ 支持二进制数据及大型对象
- ✓ 可以根据数据的特点替代 RDBMS，成为独立的数据库。或者配合 RDBMS，存储特定的数据。

### 1.3. 行式存储数据库（大数据时代）

#### 1.3.1. 行式数据库



#### 1.3.2. 列式数据库



##### 1.3.2.1. Hbase



HBase 是 **Hadoop** 项目中的数据库。它用于需要对大量的数据进行随机、实时的读写操作的场景中。

HBase 的目标就是处理数据量**非常庞大的**表，可以用**普通的计算机**处理超过 **10 亿行数据**，还可处理有数百万**列**元素的数据表。

### 1.3.2.2.Cassandra[kə'sændrə]



Apache Cassandra 是一款免费的开源 NoSQL 数据库，其设计目的在于管理由大量商用服务器构建起来的庞大集群上的**海量数据集(数据量通常达到 PB 级别)**。在众多显著特性当中，Cassandra 最为卓越的长处是对写入及读取操作进行规模调整，而且其不强调主集群的设计思路能够以相对直观的方式简化各集群的创建与扩展流程。

计算机存储单位 计算机存储单位一般用 B, KB, MB, GB, TB, EB, ZB, YB, BB 来表示，它们之间的关系是：

位 bit (比特)(Binary Digits)：存放一位二进制数，即 0 或 1，最小的存储单位。

字节 byte：8 个二进制位为一个字节(B)，最常用的单位。

1KB (Kilobyte 千字节)=1024B,

1MB (Megabyte 兆字节 简称“兆”)=1024KB,

1GB (Gigabyte 吉字节 又称“千兆”)=1024MB,

1TB (Trillionbyte 万亿字节 太字节)=1024GB, 其中  $1024=2^{10}$  (2 的 10 次方),

1PB (Petabyte 千万亿字节 拍字节)=1024TB,

1EB (Exabyte 百亿亿字节 艾字节)=1024PB,

1ZB (Zettabyte 十万万亿字节 泽字节)=1024 EB,

1YB (Jottabyte 一亿亿亿字节 尧字节)=1024 ZB,

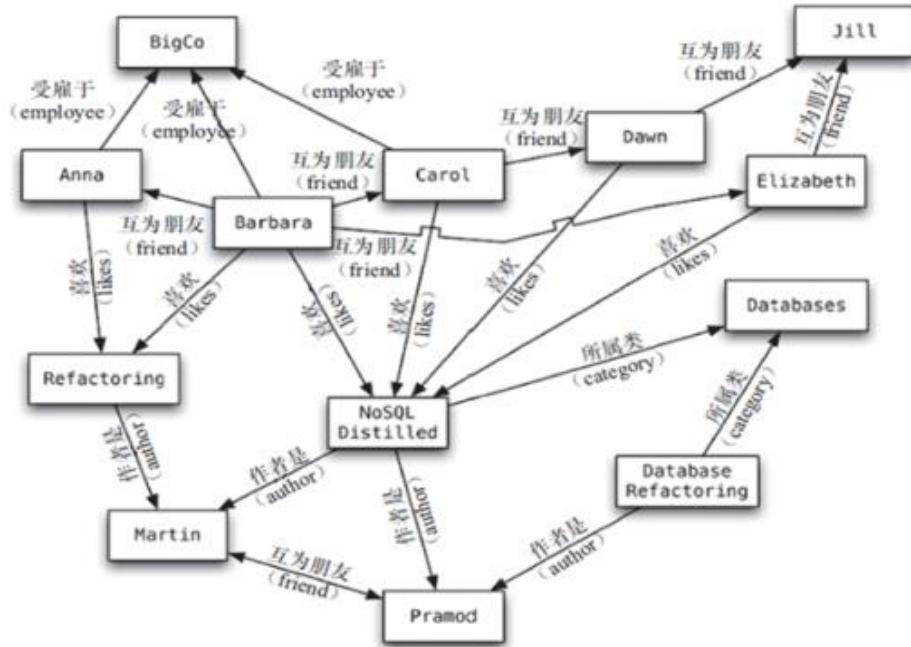
1BB (Brontobyte 一千亿亿亿字节)=1024 YB.

注：“兆”为百万级数量单位。

## 1.4. 图关系型数据库



主要应用：社会关系，公共交通网络，地图及网络拓谱( $n*(n-1)/2$ )



## 1.5. DB-Engines 数据库排名

<http://db-engines.com/en/ranking>

370 systems in ranking, April 2021

Rank	DBMS			Database Model	Score		
	Apr 2021	Mar 2021	Apr 2020		Apr 2021	Mar 2021	Apr 2020
1.	1.	1.	Oracle +	Relational, Multi-model	1274.92	-46.82	-70.51
2.	2.	2.	MySQL +	Relational, Multi-model	1220.69	-34.14	-47.66
3.	3.	3.	Microsoft SQL Server +	Relational, Multi-model	1007.97	-7.33	-75.46
4.	4.	4.	PostgreSQL +	Relational, Multi-model	553.52	+4.23	+43.66
5.	5.	5.	MongoDB +	Document, Multi-model	469.97	+7.58	+31.54
6.	6.	6.	IBM Db2 +	Relational, Multi-model	157.78	+1.77	-7.85
7.	7.	↑ 8.	Redis +	Key-value, Multi-model	155.89	+1.74	+11.08
8.	8.	↓ 7.	Elasticsearch +	Search engine, Multi-model	152.18	-0.16	+3.27
9.	9.	9.	SQLite +	Relational	125.06	+2.42	+2.87
10.	10.	10.	Microsoft Access	Relational	116.72	-1.41	-5.19
11.	11.	11.	Cassandra +	Wide column	114.85	+1.22	-5.22
12.	12.	12.	MariaDB +	Relational, Multi-model	96.37	+1.92	+6.47
13.	13.	13.	Splunk	Search engine	88.49	+1.56	+0.41
14.	14.	14.	Hive	Relational	78.50	+2.46	-5.56
15.	↑ 16.	↑ 23.	Microsoft Azure SQL Database	Relational, Multi-model	71.84	+0.96	+32.89
16.	↑ 17.	16.	Amazon DynamoDB +	Multi-model	70.73	+1.84	+6.46

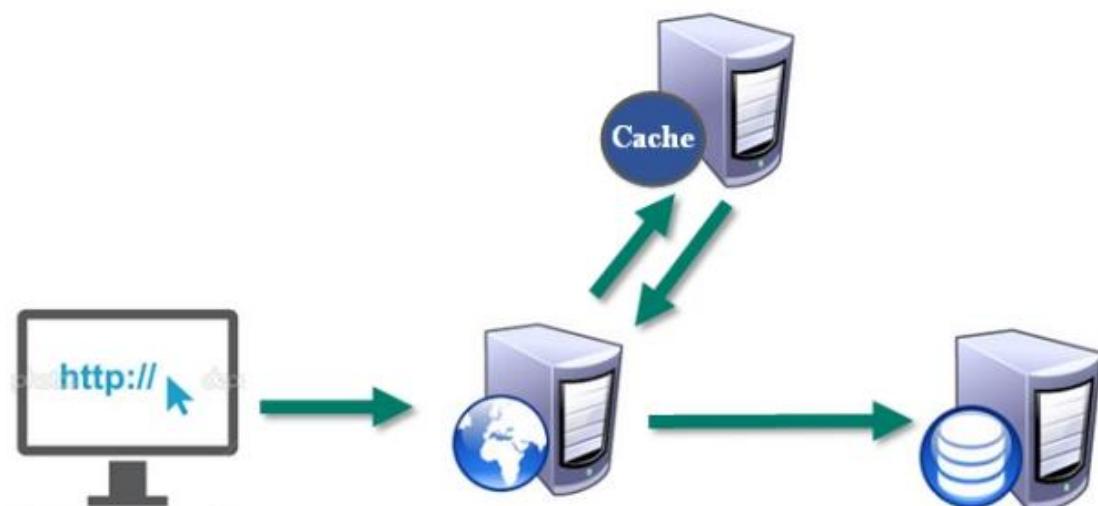
## 2. Redis 概述安装

- Redis 是一个开源的 key-value 存储系统。
- 和 Memcached 类似，它支持存储的 value 类型相对更多，包括 string(字符串)、list(链表)、set(集合)、zset(sorted set --有序集合)和 hash (哈希类型)。
- 这些数据类型都支持 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。
- 与 memcached 一样，为了保证效率，数据都是缓存在内存中。
- 区别的是 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。

### 2.1. 应用场景

#### 2.1.1. 配合关系型数据库做高速缓存

- 高频次，热门访问的数据，降低数据库 IO
- 分布式架构，做 session 共享



## 2.1.2. 多样的数据结构存储持久化数据



## 2.2. Redis 安装

Redis 官方网站	Redis 中文官方网站
<a href="http://redis.io">http://redis.io</a>	<a href="http://redis.cn/">http://redis.cn/</a>

 redis Commands Clients Documentation Community Download Modules Support Try Free

Announcing RedisConf 2021 April 20 - 21 and \$100,000 Hackathon April 15 - May 15

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 6.2.1 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](#), which are 5,000 and counting!



Redis 是一个开源 (BSD 许可) 的，内存中的数据结构存储系统，它可以用作数据库、缓存和消息中间件。它支持多种类型的数据结构，如 [字符串 \(strings\)](#)，[散列 \(hashes\)](#)，[列表 \(lists\)](#)，[集合 \(sets\)](#)，[有序集合 \(sorted sets\)](#) 与范围查询，[bitmaps](#)，[hyperloglogs](#) 和 [地理空间 \(geospatial\)](#) 索引半径查询。Redis 内置了 [复制 \(replication\)](#)，[LUA脚本 \(Lua scripting\)](#)，[LRU驱动事件 \(LRU eviction\)](#)，[事务 \(transactions\)](#) 和不同级别的 [磁盘持久化 \(persistence\)](#)，并通过 [Redis哨兵 \(Sentinel\)](#) 和自动 [分区 \(Cluster\)](#) 提供高可用性 (high availability)。

[查看Redis命令大全 →](#)

[访问Redis论坛 →](#)

[Redis使用内存计算器 →](#)

### 试用 (Try it)

准备使用Redis? 通过 [互动教程 \(interactive tutorial\)](#) 将引导您了解 Redis 最重要的特征。

### 下载 (Download it)

最新稳定版本是 redis 6.0.6 想了解更多候选版本或者测试版本? [点击这里查看更多。](#)

### 链接 (Quick links)

在 Twitter 和 GitHub 查看与 Redis 同步更新的更多资料。通过订阅 [我们的邮件列表](#) 获得帮助或者帮助其他人，现在订阅人数已经超过 5000 人，并且还在不断增加哦。

## 2.2.1. 安装版本

- 6.2.1 for Linux ([redis-6.2.1.tar.gz](#))
- 不用考虑在 windows 环境下对 Redis 的支持

### Windows

The Redis project does not officially support Windows. However, the Microsoft Open Tech group develops and maintains this Windows port targeting Win64. Learn more.

## 2.2.2. 安装步骤

### 2.2.2.1. 准备工作：下载安装最新版的 gcc 编译器

安装 C 语言的编译环境

```
yum install centos-release-scl scl-utils-build
```

```
yum install -y devtoolset-8-toolchain
```

```
scl enable devtoolset-8 bash
```

测试 gcc 版本

```
gcc --version
```

```
[bigdata@hdp1 redis-6.0.8]$ sudo gcc --version
gcc (GCC) 8.3.1 20190311 (Red Hat 8.3.1-3)
Copyright (C) 2018 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

2.2.2.2. 下载 redis-6.2.1.tar.gz 放 /opt 目录

2.2.2.3. 解压命令: tar -zxvf redis-6.2.1.tar.gz

2.2.2.4. 解压完成后进入目录: cd redis-6.2.1

2.2.2.5. 在 redis-6.2.1 目录下再次执行 make 命令 (只是编译好)

2.2.2.6. 如果没有准备好 C 语言编译环境, make 会报错  
—Jemalloc/jemalloc.h: 没有那个文件

```
[root@zy redis-3.2.5]# make
cd src && make all
make[1]: 进入目录"/opt/redis-3.2.5/src"
  CC adlist.o
In file included from adlist.c:34:0:
zmalloc.h:50:31: 致命错误: jemalloc/jemalloc.h: 没有那个文件或目录
 #include <jemalloc/jemalloc.h>
^
编译中断。
make[1]: *** [adlist.o] 错误 1
make[1]: 离开目录"/opt/redis-3.2.5/src"
make: *** [all] 错误 2
```

### 2.2.2.7. 解决方案：运行 make distclean

```
[root@zy redis-3.2.5]# make distclean
cd src && make distclean
make[1]: 进入目录"/opt/redis-3.2.5/src"
rm -rf redis-server redis-sentinel redis-cli redis-benchmark redis-check-rdb redis-check-aof *.o *.gcda *.gcno *.gcov redis.info lcov-html
(cd ..deps && make distclean)
make[2]: 进入目录"/opt/redis-3.2.5/deps"
(cd hiredis && make clean) > /dev/null || true
(cd linenoise && make clean) > /dev/null || true
(cd lua && make clean) > /dev/null || true
(cd geohash-int && make clean) > /dev/null || true
(cd jemalloc && [ -f Makefile ] && make distclean) > /dev/null || true
(rm -f .make-*)
make[2]: 离开目录"/opt/redis-3.2.5/deps"
(rm -f .make-*)
make[1]: 离开目录"/opt/redis-3.2.5/src"
```

### 2.2.2.8. 在 redis-6.2.1 目录下再次执行 make 命令（只是编译好）

```
CC latency.o
CC sparkline.o
CC redis-check-rdb.o
CC geo.o
LINK redis-server
INSTALL redis-sentinel
CC redis-cli.o
LINK redis-cli
CC redis-benchmark.o
LINK redis-benchmark
INSTALL redis-check-rdb
CC redis-check-aof.o
LINK redis-check-aof

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

make[1]: 离开目录"/opt/redis-3.2.5/src"
```

### 2.2.2.9. 跳过 make test 继续执行：make install

```
[root@zy redis-3.2.5]# make install
cd src && make install
make[1]: 进入目录"/opt/redis-3.2.5/src"

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

INSTALL install
INSTALL install
INSTALL install
INSTALL install
INSTALL install
make[1]: 离开目录"/opt/redis-3.2.5/src"
```

### 2.2.3. 安装目录: /usr/local/bin

查看默认安装目录:

redis-benchmark: 性能测试工具, 可以在自己本子运行, 看看自己本子性能如何

redis-check-aof: 修复有问题的 AOF 文件, rdb 和 aof 后面讲

redis-check-dump: 修复有问题的 dump.rdb 文件

redis-sentinel: Redis 集群使用

redis-server: Redis 服务器启动命令

redis-cli: 客户端, 操作入口

### 2.2.4. 前台启动 (不推荐)

前台启动, 命令行窗口不能关闭, 否则服务器停止

```
[root@zyz bin]# redis-server
12471:C 16 Dec 12:33:08.178 # Warning: no config file specified, using the default config. In order to specify a config file use redis-server /path/to/redis.conf
12471:M 16 Dec 12:33:08.178 * Increased maximum number of open files to 10032 (it was originally set to 1024).

                               Redis 3.2.5 (00000000/0) 64 bit
                               Running in standalone mode
                               Port: 6379
                               PID: 12471

                               http://redis.io

12471:M 16 Dec 12:33:08.181 # WARNING: The TCP backlog setting of 511 cannot be enforced because /proc/sys/net/core/somaxconn is set to the lower value of 128.
12471:M 16 Dec 12:33:08.181 # Server started, Redis version 3.2.5
12471:M 16 Dec 12:33:08.181 # 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.
12471:M 16 Dec 12:33:08.182 # WARNING you have Transparent Huge Pages (THP) support enabled in your kernel. This will create latency and memory usage issues with Redis. To fix this issue run the command 'echo never > /sys/kernel/mm/transparent_hugepage/enabled' as root, and add it to your /etc/rc.local in order to retain the setting after a reboot. Redis must be restarted after THP is disabled.
12471:M 16 Dec 12:33:08.182 * The server is now ready to accept connections on port 6379
```

## 2.2.5. 后台启动（推荐）

### 2.2.5.1. 备份 redis.conf

拷贝一份 redis.conf 到其他目录

```
cp /opt/redis-3.2.5/redis.conf /myredis
```

### 2.2.5.2. 后台启动设置 daemonize no 改成 yes

修改 redis.conf(128 行)文件将里面的 daemonize no 改成 yes，让服务在后台启动

### 2.2.5.3. Redis 启动

redis-server/myredis/redis.conf

```
[root@zy bin]# redis-server /myredis/redis.conf
[root@zy bin]# ps -ef|grep redis
root      12639     1  0 12:41 ?        00:00:00 redis-server 127.0.0.1:6379
root      12661   6326  0 12:42 pts/2    00:00:00 grep --color=auto redis
```

### 2.2.5.4. 用客户端访问：redis-cli

```
[root@zy bin]# redis-cli
127.0.0.1:6379> 
```

### 2.2.5.5. 多个端口可以：redis-cli -p6379

### 2.2.5.6. 测试验证：ping

```
127.0.0.1:6379> ping
PONG
```

### 2.2.5.7. Redis 关闭

单实例关闭：redis-cli shutdown

```
[root@zy bin]# redis-server /myredis/redis.conf
[root@zy bin]# redis-cli shutdown
[root@zy bin]# ps -ef|grep redis
root      12722   6326  0 12:47 pts/2    00:00:00 grep --color=auto redis
```

也可以进入终端后再关闭

```
127.0.0.1:6379> shutdown  
not connected>
```

多实例关闭，指定端口关闭：redis-cli -p 6379 shutdown

## 2.2.6. Redis 介绍相关知识

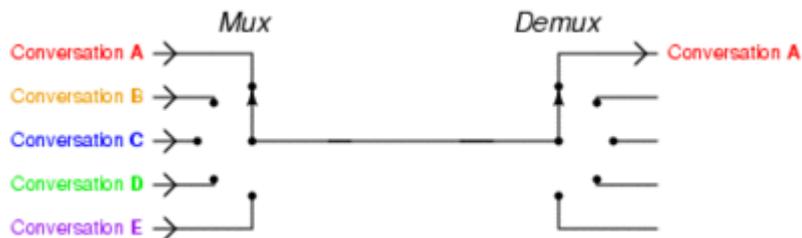
端口 <b>6379</b> 从何而来  Alessia Merz 	默认 16 个数据库，类似数组下标从 0 开始，初始 <b>默认使用 0 号库</b>  使用命令 select <dbid> 来切换数据库。如: select 8  统一密码管理，所有库同样密码。  <b>dbsize</b> 查看当前数据库的 key 的数量  <b>flushdb</b> 清空当前库  <b>flushall</b> 通杀全部库
--	--

Redis 是单线程+多路 IO 复用技术

多路复用是指使用一个线程来检查多个文件描述符（Socket）的就绪状态，比如调用 select 和 poll 函数，传入多个文件描述符，如果有一个文件描述符就绪，则返回，否则阻塞直到超时。得到就绪状态后进行真正的操作可以在同一个线程里执行，也可以启动线程执行（比如使用线程池）

**串行 vs 多线程+锁 (memcached) vs 单线程+多路 IO 复用(Redis)**

(与 Memcache 三点不同: 支持多数据类型，支持持久化，单线程+多路 IO 复用)



## 3. 常用五大数据类型

哪里去获得 redis 常见数据类型操作命令 <http://www.redis.cn/commands.html>

### 3.1. Redis 键(key)

keys \* 查看当前库所有 key (匹配: keys \*1)

exists key 判断某个 key 是否存在

type key 查看你的 key 是什么类型

del key 删除指定的 key 数据

**unlink key** 根据 value 选择非阻塞删除

仅将 keys 从 keyspace 元数据中删除，真正的删除会在后续异步操作。

expire key 10 10 秒钟：为给定的 key 设置过期时间

ttl key 查看还有多少秒过期，-1 表示永不过期，-2 表示已过期

select 命令切换数据库

dbsize 查看当前数据库的 key 的数量

flushdb 清空当前库

flushall 通杀全部库

### 3.2. Redis 字符串(String)

#### 3.2.1. 简介

String 是 Redis 最基本的类型，你可以理解成与 Memcached 一模一样的类型，一个 key 对应一个 value。

String 类型是**二进制安全的**。意味着 Redis 的 string 可以包含任何数据。比如 jpg 图片或者序列化的对象。

String 类型是 Redis 最基本的数据类型，一个 Redis 中字符串 value 最多可以是 **512M**

#### 3.2.2. 常用命令

set <key><value>添加键值对

```
127.0.0.1:6379> set key value [EX seconds|PX milliseconds|KEEPTTL] [NX|XX]
```

\*NX: 当数据库中 key 不存在时, 可以将 key-value 添加数据库

\*XX: 当数据库中 key 存在时, 可以将 key-value 添加数据库, 与 NX 参数互斥

\*EX: key 的超时秒数

\*PX: key 的超时毫秒数, 与 EX 互斥

get <key>查询对应键值

append <key><value>将给定的<value>追加到原值的末尾

strlen <key>获得值的长度

setnx <key><value>只有在 key 不存在时 设置 key 的值

incr <key>

将 key 中储存的数字值增 1

只能对数字值操作, 如果为空, 新增加值为 1

decr <key>

将 key 中储存的数字值减 1

只能对数字值操作, 如果为空, 新增加值为-1

incrby / decrby <key><步长>将 key 中储存的数字值增减。自定义步长。

### 原子性

#### INCR key

起始版本: 1.0.0

时间复杂度: O(1)

对存储在指定key的数值执行原子的加1操作。

所谓原子操作是指不会被线程调度机制打断的操作;

这种操作一旦开始, 就一直运行到结束, 中间不会有任何 context switch (切换到另

一个线程)。

(1) 在单线程中，能够在单条指令中完成的操作都可以认为是"原子操作"，因为中断只能发生于指令之间。

(2) 在多线程中，不能被其它进程（线程）打断的操作就叫原子操作。

Redis 单命令的原子性主要得益于 Redis 的单线程。

**案例：**

java 中的 i++是否是原子操作？

i=0;两个线程分别对 i 进行++100 次,值是多少?

i=0	i=0
i++	
i=99	i++
	i=1
i=1	i++
	i=100
i++	
i=2	

mset <key1><value1><key2><value2> .....

同时设置一个或多个 key-value 对

mget <key1><key2><key3> .....

同时获取一个或多个 value

msetnx <key1><value1><key2><value2> .....

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

**原子性，有一个失败则都失败**

getrange <key><起始位置><结束位置>

获得值的范围，类似 java 中的 substring，**前包，后包**

setrange <key><起始位置><value>

用 <value> 覆写<key>所储存的字符串值，从<起始位置>开始(**索引从 0 开始**)。

**setex <key><过期时间><value>**

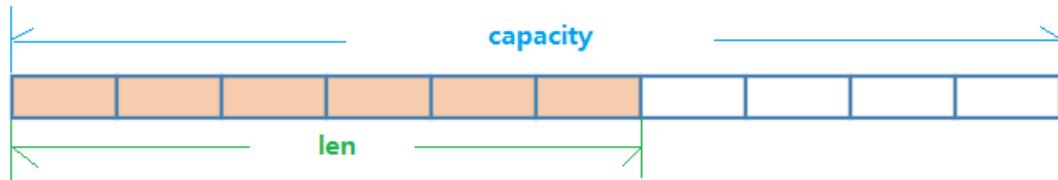
设置键值的同时，设置过期时间，单位秒。

getset <key><value>

以新换旧，设置了新值同时获得旧值。

### 3.2.3. 数据结构

String 的数据结构为简单动态字符串(Simple Dynamic String, 缩写 SDS)。是可以修改的字符串，内部结构实现上类似于 Java 的 ArrayList，采用预分配冗余空间的方式来减少内存的频繁分配。



如图中所示，内部为当前字符串实际分配的空间 capacity 一般要高于实际字符串长度 len。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，扩容时一次只会多扩 1M 的空间。需要注意的是字符串最大长度为 512M。

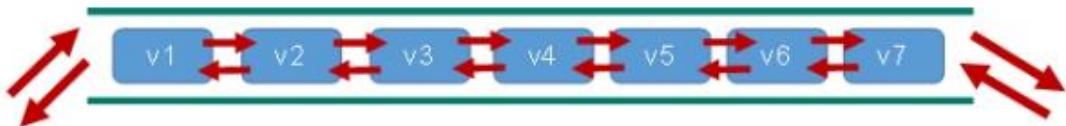
## 3.3. Redis 列表(List)

### 3.3.1. 简介

单键多值

Redis 列表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）。

它的底层实际是个**双向链表**，对两端的操作性能很高，通过索引下标的操作中间的节点性能会较差。



### 3.3.2. 常用命令

lpush/rpush <key><value1><value2><value3> .... 从左边/右边插入一个或多个值。

lpop/rpop <key> 从左边/右边吐出一个值。值在键在，值光键亡。

rpoplpush <key1><key2> 从<key1>列表右边吐出一个值，插到<key2>列表左边。

lrange <key><start><stop>

按照索引下标获得元素(从左到右)

lrange mylist 0 -1 0 左边第一个， -1 右边第一个， (0-1 表示获取所有)

lindex <key><index> 按照索引下标获得元素(从左到右)

llen <key> 获得列表长度

linsert <key> before <value><newvalue> 在<value>的后面插入<newvalue>插入值

lrem <key><n><value> 从左边删除 n 个 value(从左到右)

lset<key><index><value> 将列表 key 下标为 index 的值替换成 value

### 3.3.3. 数据结构

List 的数据结构为快速链表 quickList。

首先在列表元素较少的情况下会使用一块连续的内存存储，这个结构是 ziplist，也即是压缩列表。

它将所有的元素紧挨着一起存储，分配的是一块连续的内存。

当数据量比较多的时候才会改成 quicklist。

因为普通的链表需要的附加指针空间太大，会比较浪费空间。比如这个列表里存的只是 int 类型的数据，结构上还需要两个额外的指针 prev 和 next。



Redis 将链表和 ziplist 结合起来组成了 quicklist。也就是将多个 ziplist 使用双向指针串起来使用。这样既满足了快速的插入删除性能，又不会出现太大的空间冗余。

## 3.4. Redis 集合(Set)

### 3.4.1. 简介

Redis set 对外提供的功能与 list 类似是一个列表的功能，特殊之处在于 set 是可以**自动排重**的，当你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，并且 set 提供了判断某个成员是否在一个 set 集合内的重要接口，这个也是 list 所不能提供的。

Redis 的 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**复杂度都是 O(1)**。

一个算法，随着数据的增加，执行时间的长短，如果是 O(1)，数据增加，查找数据的时间不变

### 3.4.2. 常用命令

sadd <key><value1><value2> .....

将一个或多个 member 元素加入到集合 key 中，已经存在的 member 元素将被忽略

smembers <key> 取出该集合的所有值。

sismember <key><value> 判断集合<key>是否为含有该<value>值，有 1，没有 0

scard<key> 返回该集合的元素个数。

srem <key><value1><value2> .... 删除集合中的某个元素。

spop <key>随机从该集合中吐出一个值。

srandmember <key><n>随机从该集合中取出 n 个值。不会从集合中删除。

smove <source><destination>value 把集合中一个值从一个集合移动到另一个集合

sinter <key1><key2>返回两个集合的交集元素。

sunion <key1><key2>返回两个集合的并集元素。

sdiff <key1><key2>返回两个集合的差集元素(key1 中的，不包含 key2 中的)

### 3.4.3. 数据结构

Set 数据结构是 dict 字典，字典是用哈希表实现的。

Java 中 HashSet 的内部实现使用的是 HashMap，只不过所有的 value 都指向同一个对象。Redis 的 set 结构也是一样，它的内部也使用 hash 结构，所有的 value 都指向同一个内部值。

## 3.5. Redis 哈希(Hash)

### 3.5.1. 简介

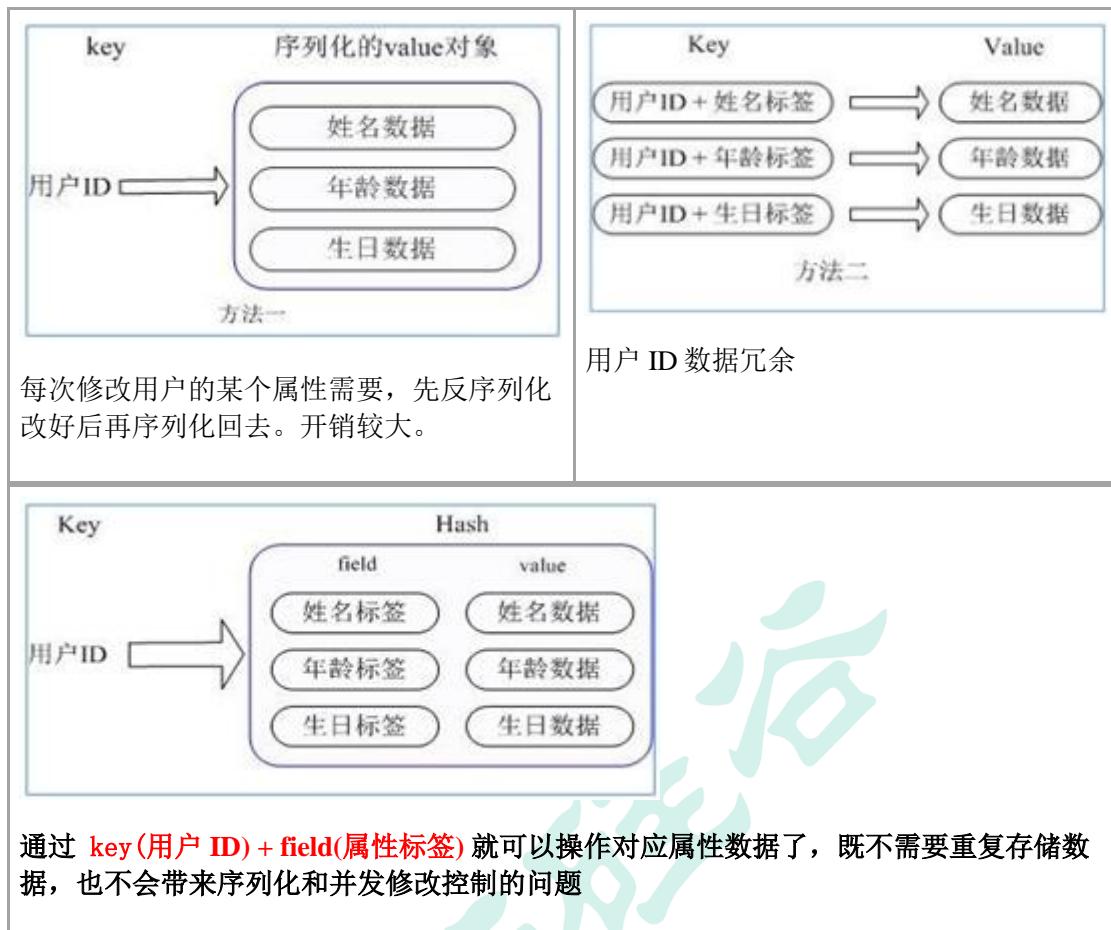
Redis hash 是一个键值对集合。

Redis hash 是一个 string 类型的 field 和 value 的映射表，hash 特别适合用于存储对象。

类似 Java 里面的 Map<String, Object>

用户 ID 为查找的 key，存储的 value 用户对象包含姓名，年龄，生日等信息，如果用普通的 key/value 结构来存储

主要有以下 2 种存储方式：



### 3.5.2. 常用命令

hset <key><field><value>给<key>集合中的 <field>键赋值<value>

hget <key1><field>从<key1>集合<field>取出 value

hmset <key1><field1><value1><field2><value2>... 批量设置 hash 的值

hexists<key1><field>查看哈希表 key 中，给定域 field 是否存在。

hkeys <key>列出该 hash 集合的所有 field

hvals <key>列出该 hash 集合的所有 value

hincrby <key><field><increment>为哈希表 key 中的域 field 的值加上增量 1 -1

hsetnx <key><field><value>将哈希表 key 中的域 field 的值设置为 value，当且仅当域 field 不存在 .

### 3.5.3. 数据结构

Hash 类型对应的数据结构是两种：ziplist（压缩列表），hashtable（哈希表）。当 field-value 长度较短且个数较少时，使用 ziplist，否则使用 hashtable。

## 3.6. Redis 有序集合 Zset(sorted set)

### 3.6.1. 简介

Redis 有序集合 zset 与普通集合 set 非常相似，是一个**没有重复元素**的字符串集合。

不同之处是有序集合的每个成员都关联了一个**评分 (score)**，这个评分 (score) 被用来按照从最低分到最高分的方式排序集合中的成员。**集合的成员是唯一的，但是评分可以是重复了。**

因为元素是有序的，所以你也可以很快的根据评分 (score) 或者次序 (position) 来获取一个范围的元素。

访问有序集合的中间元素也是非常快的，因此你能够使用有序集合作为一个没有重复成员的智能列表。

### 3.6.2. 常用命令

**zadd <key><score1><value1><score2><value2>...**

将一个或多个 member 元素及其 score 值加入到有序集 key 当中。

**zrange <key><start><stop> [WITHSCORES]**

返回有序集 key 中，下标在<start><stop>之间的元素

带 WITHSCORES，可以让分数一起和值返回到结果集。

**zrangebyscore key minmax [withscores] [limit offset count]**

返回有序集 key 中，所有 score 值介于 min 和 max 之间(包括等于 min 或 max )的成员。有序集成员按 score 值递增(从小到大)次序排列。

**zrevrangebyscore key maxmin [withscores] [limit offset count]**

同上，改为从大到小排列。

zincrby <key><increment><value> 为元素的 score 加上增量

zrem <key><value> 删除该集合下，指定值的元素

zcount <key><min><max> 统计该集合，分数区间内的元素个数

zrank <key><value> 返回该值在集合中的排名，从 0 开始。

案例：如何利用 zset 实现一个文章访问量的排行榜？

```
127.0.0.1:6379> zadd topn 1000 v1 2000 v2 3000 v3  
(integer) 3  
127.0.0.1:6379> zrevrange topn 0 9 withscores  
1) "v3"  
2) "3000"  
3) "v2"  
4) "2000"  
5) "v1"  
6) "1000"
```

### 3.6.3. 数据结构

SortedSet(zset)是Redis提供的一个非常特别的数据结构，一方面它等价于Java的数据结构 Map<String, Double>，可以给每一个元素 value 赋予一个权重 score，另一方面它又类似于 TreeSet，内部的元素会按照权重 score 进行排序，可以得到每个元素的名次，还可以通过 score 的范围来获取元素的列表。

zset 底层使用了两个数据结构

(1) hash, hash 的作用就是关联元素 value 和权重 score，保障元素 value 的唯一性，可以通过元素 value 找到相应的 score 值。

(2) 跳跃表，跳跃表的目的在于给元素 value 排序，根据 score 的范围获取元素列表。

### 3.6.4. 跳跃表（跳表）

#### 1、简介

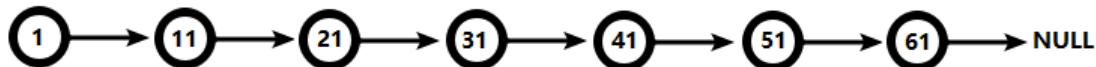
有序集合在生活中比较常见，例如根据成绩对学生排名，根据得分对玩家排名等。对于有序集合的底层实现，可以用数组、平衡树、链表等。数组不便元素的插入、

删除；平衡树或红黑树虽然效率高但结构复杂；链表查询需要遍历所有效率低。Redis 采用的是跳跃表。跳跃表效率堪比红黑树，实现远比红黑树简单。

## 2、实例

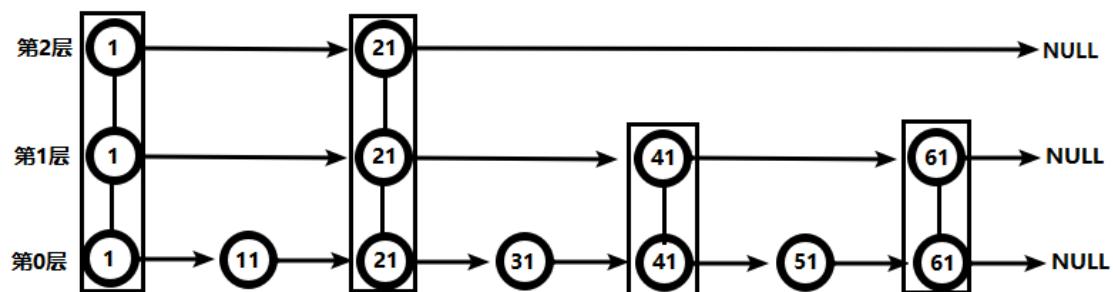
对比有序链表和跳跃表，从链表中查询出 51

### (1) 有序链表



要查找值为 51 的元素，需要从第一个元素开始依次查找、比较才能找到。共需要 6 次比较。

### (2) 跳跃表



从第 2 层开始，1 节点比 51 节点小，向后比较。

21 节点比 51 节点小，继续向后比较，后面就是 NULL 了，所以从 21 节点向下到第 1 层在第 1 层，41 节点比 51 节点小，继续向后，61 节点比 51 节点大，所以从 41 向下在第 0 层，51 节点为要查找的节点，节点被找到，共查找 4 次。

从此可以看出跳跃表比有序链表效率要高

## 4. Redis 配置文件介绍

自定义目录：/myredis/redis.conf

## 4.1. ###Units 单位###

配置大小单位,开头定义了一些基本的度量单位, 只支持 bytes, 不支持 bit

大小写不敏感

```
1 # Redis configuration file example.
2 #
3 # Note that in order to read the configuration file, Redis must be
4 # started with the file path as first argument:
5 #
6 # ./redis-server /path/to/redis.conf
7
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.
```

## 4.2. ###INCLUDES 包含###

```
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 # Notice 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
```

类似 jsp 中的 include, 多实例的情况下可以把公用的配置文件提取出来

## 4.3. ###网络相关配置 ###

### 4.3.1. bind

默认情况 bind=127.0.0.1 只能接受本机的访问请求

不写的情况下, 无限制接受任何 ip 地址的访问

生产环境肯定要写你应用服务器的地址；服务器是需要远程访问的，所以需要将其注释掉

如果开启了 protected-mode，那么在没有设定 bind ip 且没有设密码的情况下，Redis 只允许接受本机的响应

```
40 # By default, if no "bind" configuration directive is specified, Redis listens
41 # for connections from all the network interfaces available on the server.
42 # It is possible to listen to just one or multiple selected interfaces using
43 # the "bind" configuration directive, followed by one or more IP addresses.
44 #
45 # Examples:
46 #
47 # bind 192.168.1.100 10.0.0.1
48 # bind 127.0.0.1 ::1
49 #
50 # ~~~ WARNING ~~~ If the computer running Redis is directly exposed to the
51 # internet, binding to all the interfaces is dangerous and will expose the
52 # instance to everybody on the internet. So by default we uncomment the
53 # following bind directive, that will force Redis to listen only into
54 # the IPv4 lookback interface address (this means Redis will be able to
55 # accept connections only from clients running into the same computer it
56 # is running).
57 #
58 # IF YOU ARE SURE YOU WANT YOUR INSTANCE TO LISTEN TO ALL THE INTERFACES
59 # JUST COMMENT THE FOLLOWING LINE.
60 #
61 #bind 127.0.0.1
```

保存配置，停止服务，重启启动查看进程，不再是本机访问了。

```
[root@zy myredis]# redis-cli
127.0.0.1:6379> shutdown
not connected>
[root@zy myredis]# redis-server /myredis/redis.conf
[root@zy myredis]# ps -ef|grep redis
root      13549      1  0 16:23 ?        00:00:00 redis-server *:6379
root      13553  12006  0 16:23 pts/2    00:00:00 grep --color=auto redis
```

### 4.3.2. protected-mode

将本机访问保护模式设置 no

```
63 # Protected mode is a layer of security protection, in order to avoid that
64 # Redis instances left open on the internet are accessed and exploited.
65 #
66 # When protected mode is on and if:
67 #
68 # 1) The server is not binding explicitly to a set of addresses using the
69 #    "bind" directive.
70 # 2) No password is configured.
71 #
72 # The server only accepts connections from clients connecting from the
73 # IPv4 and IPv6 loopback addresses 127.0.0.1 and ::1, and from Unix domain
74 # sockets.
75 #
76 # By default protected mode is enabled. You should disable it only if
77 # you are sure you want clients from other hosts to connect to Redis
78 # even if no authentication is configured, nor a specific set of interfaces
79 # are explicitly listed using the "bind" directive.
80 protected-mode no
```

### 4.3.3. Port

端口号， 默认 6379

```
82 # Accept connections on the specified port, default is 6379 (IANA #815344).
83 # If port 0 is specified Redis will not listen on a TCP socket.
84 port 6379
```

### 4.3.4. tcp-backlog

设置 tcp 的 backlog， backlog 其实是一个连接队列， backlog 队列总和=未完成三次握手队列 + 已经完成三次握手队列。

在高并发环境下你需要一个高 backlog 值来避免慢客户端连接问题。

注意 Linux 内核会将这个值减小到 /proc/sys/net/core/somaxconn 的值 (128)， 所以需要确认增大 /proc/sys/net/core/somaxconn 和 /proc/sys/net/ipv4/tcp\_max\_syn\_backlog (128) 两个值来达到想要的效果

```
86 # TCP listen() backlog.
87 #
88 # In high requests-per-second environments you need an high backlog in order
89 # to avoid slow clients connections issues. Note that the Linux kernel
90 # will silently truncate it to the value of /proc/sys/net/core/somaxconn so
91 # make sure to raise both the value of somaxconn and tcp_max_syn_backlog
92 # in order to get the desired effect.
93 tcp-backlog 511
```

### 4.3.5. timeout

一个空闲的客户端维持多少秒会关闭， 0 表示关闭该功能。即永不关闭。

```
95 # Unix socket.
96 #
97 # Specify the path for the Unix socket that will be used to listen for
98 # incoming connections. There is no default, so Redis will not listen
99 # on a unix socket when not specified.
100 #
101 # unixsocket /tmp/redis.sock
102 # unixsocketperm 700
103
104 # Close the connection after a client is idle for N seconds (0 to disable)
105 timeout 0
```

#### 4.3.6. tcp-keepalive

对访问客户端的一种心跳检测，每个 n 秒检测一次。

单位为秒，如果设置为 0，则不会进行 Keepalive 检测，建议设置成 60

```
107 # TCP keepalive.
108 #
109 # If non-zero, use SO_KEEPALIVE to send TCP ACKs to clients in absence
110 # of communication. This is useful for two reasons:
111 #
112 # 1) Detect dead peers.
113 # 2) Take the connection alive from the point of view of network
114 #     equipment in the middle.
115 #
116 # On Linux, the specified value (in seconds) is the period used to send ACKs.
117 # Note that to close the connection the double of the time is needed.
118 # On other kernels the period depends on the kernel configuration.
119 #
120 # A reasonable value for this option is 300 seconds, which is the new
121 # Redis default starting with Redis 3.2.1.
122 tcp-keepalive 300
```

### 4.4. ###GENERAL 通用###

#### 4.4.1. daemonize

是否为后台进程，设置为 yes

守护进程，后台启动

```
126 # By default Redis does not run as a daemon. Use 'yes' if you need it.
127 # Note that Redis will write a pid file in /var/run/redis.pid when daemonized.
128 daemonize yes
```

#### 4.4.2. pidfile

存放 pid 文件的位置，每个实例会产生一个不同的 pid 文件

```
141 # If a pid file is specified, Redis writes it where specified at startup
142 # and removes it at exit.
143 #
144 # When the server runs non daemonized, no pid file is created if none is
145 # specified in the configuration. When the server is daemonized, the pid file
146 # is used even if not specified, defaulting to "/var/run/redis.pid".
147 #
148 # Creating a pid file is best effort: if Redis is not able to create it
149 # nothing bad happens, the server will start and run normally.
150 pidfile /var/run/redis_6379.pid
```

#### 4.4.3. loglevel

指定日志记录级别，Redis 总共支持四个级别：debug、verbose、notice、warning，默认为 notice

四个级别根据使用阶段来选择，生产环境选择 notice 或者 warning

```
152 # Specify the server verbosity level.
153 # This can be one of:
154 # debug (a lot of information, useful for development/testing)
155 # verbose (many rarely useful info, but not a mess like the debug level)
156 # notice (moderately verbose, what you want in production probably)
157 # warning (only very important / critical messages are logged)
158 loglevel notice
```

#### 4.4.4. logfile

日志文件名称

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

#### 4.4.5. databases 16

设定库的数量 默认 16，默认数据库为 0，可以使用 SELECT <dbid>命令在连接上指定数据库 id

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

## 4.5. ###SECURITY 安全###

### 4.5.1. 设置密码

```
467 ##### SECURITY #####
468
469 # Require clients to issue AUTH <PASSWORD> before processing any other
470 # commands. This might be useful in environments in which you do not trust
471 # others with access to the host running redis-server.
472 #
473 # This should stay commented out for backward compatibility and because most
474 # people do not need auth (e.g. they run their own servers).
475 #
476 # Warning: since Redis is pretty fast an outside user can try up to
477 # 150k passwords per second against a good box. This means that you should
478 # use a very strong password otherwise it will be very easy to break.
479 #
480 # requirepass foobared
```

访问密码的查看、设置和取消

在命令中设置密码，只是临时的。重启 redis 服务器，密码就还原了。

永久设置，需要再配置文件中进行设置。

```
127.0.0.1:6379> config get requirepass
1) "requirepass"
2)

127.0.0.1:6379> config set requirepass "123456"
OK

127.0.0.1:6379> config get requirepass
(error) NOAUTH Authentication required.
127.0.0.1:6379> auth 123456
OK

127.0.0.1:6379> get k1
"v1"

127.0.0.1:6379> config set requirepass ""
OK

127.0.0.1:6379> config get requirepass
1) "requirepass"
2)

127.0.0.1:6379>
```

## 4.6. ##### LIMITS 限制 ####

### 4.6.1. maxclients

- 设置 redis 同时可以与多少个客户端进行连接。

- 默认情况下为 10000 个客户端。
- 如果达到了此限制，redis 则会拒绝新的连接请求，并且向这些连接请求方发出“max number of clients reached”以作回应。

```
503 # Set the max number of connected clients at the same time. By default
504 # this limit is set to 10000 clients, however if the Redis server is not
505 # able to configure the process file limit to allow for the specified limit
506 # the max number of allowed clients is set to the current file limit
507 # minus 32 (as Redis reserves a few file descriptors for internal uses).
508 #
509 # Once the limit is reached Redis will close all the new connections sending
510 # an error 'max number of clients reached'.
511 #
512 # maxclients 10000
```

#### 4.6.2. maxmemory

- 建议**必须设置**，否则，将内存占满，造成服务器宕机
- 设置 redis 可以使用的内存量。一旦到达内存使用上限，redis 将会试图移除内部数据，移除规则可以通过 **maxmemory-policy** 来指定。
- 如果 redis 无法根据移除规则来移除内存中的数据，或者设置了“不允许移除”，那么 redis 则会针对那些需要申请内存的指令返回错误信息，比如 SET、LPUSH 等。
- 但是对于无内存申请的指令，仍然会正常响应，比如 GET 等。如果你的 redis 是主 redis（说明你的 redis 有从 redis），那么在设置内存使用上限时，需要在系统中留出一些内存空间给同步队列缓存，只有在你设置的是“不移除”的情况下，才不用考虑这个因素。

```
514 # Don't use more memory than the specified amount of bytes.
515 # When the memory limit is reached Redis will try to remove keys
516 # according to the eviction policy selected (see maxmemory-policy).
517 #
518 # If Redis can't remove keys according to the policy, or if the policy is
519 # set to 'noeviction', Redis will start to reply with errors to commands
520 # that would use more memory, like SET, LPUSH, and so on, and will continue
521 # to reply to read-only commands like GET.
522 #
523 # This option is usually useful when using Redis as an LRU cache, or to set
524 # a hard memory limit for an instance (using the 'noeviction' policy).
525 #
526 # WARNING: If you have slaves attached to an instance with maxmemory on,
527 # the size of the output buffers needed to feed the slaves are subtracted
528 # from the used memory count, so that network problems / resyncs will
529 # not trigger a loop where keys are evicted, and in turn the output
530 # buffer of slaves is full with DELs of keys evicted triggering the deletion
531 # of more keys, and so forth until the database is completely emptied.
532 #
533 # In short... if you have slaves attached it is suggested that you set a lower
534 # limit for maxmemory so that there is some free RAM on the system for slave
535 # output buffers (but this is not needed if the policy is 'noeviction').
536 #
537 # maxmemory <bytes>
```

### 4.6.3. maxmemory-policy

- volatile-lru: 使用 LRU 算法移除 key, 只对设置了过期时间的键; (最近最少使用)
- allkeys-lru: 在所有集合 key 中, 使用 LRU 算法移除 key
- volatile-random: 在过期集合中移除随机的 key, 只对设置了过期时间的键
- allkeys-random: 在所有集合 key 中, 移除随机的 key
- volatile-ttl: 移除那些 TTL 值最小的 key, 即那些最近要过期的 key
- noeviction: 不进行移除。针对写操作, 只是返回错误信息

```
539 # MAXMEMORY POLICY: how Redis will select what to remove when maxmemory
540 # is reached. You can select among five behaviors:
541 #
542 # volatile-lru -> remove the key with an expire set using an LRU algorithm
543 # allkeys-lru -> remove any key according to the LRU algorithm
544 # volatile-random -> remove a random key with an expire set
545 # allkeys-random -> remove a random key, any key
546 # volatile-ttl -> remove the key with the nearest expire time (minor TTL)
547 # noeviction -> don't expire at all, just return an error on write operations
548 #
549 # Note: with any of the above policies, Redis will return an error on write
550 #       operations, when there are no suitable keys for eviction.
551 #
552 #       At the date of writing these commands are: set setnx setex append
553 #           incr decr rpush lpush rpushx lpushx linsert lset rpoplpush sadd
554 #           sinter sinterstore sunion sunionstore sdiff sdifflist zadd zincrby
555 #           zunionstore zinterstore hset hsetnx hmset hincrby incrby decrby
556 #           getset mset msetnx exec sort
557 #
558 # The default is:
559 #
560 # maxmemory-policy noeviction
```

### 4.6.4. maxmemory-samples

- 设置样本数量, LRU 算法和最小 TTL 算法都并非是精确的算法, 而是估算值, 所以你可以设置样本的大小, redis 默认会检查这么多个 key 并选择其中 LRU 的那个。
- 一般设置 3 到 7 的数字, 数值越小样本越不准确, 但性能消耗越小。

```
562 # LRU and minimal TTL algorithms are not precise algorithms but approximated
563 # algorithms (in order to save memory), so you can tune it for speed or
564 # accuracy. For default Redis will check five keys and pick the one that was
565 # used less recently, you can change the sample size using the following
566 # configuration directive.
567 #
568 # The default of 5 produces good enough results. 10 Approximates very closely
569 # true LRU but costs a bit more CPU. 3 is very fast but not very accurate.
570 #
571 # maxmemory-samples 5
```

## 5. Redis 的发布和订阅

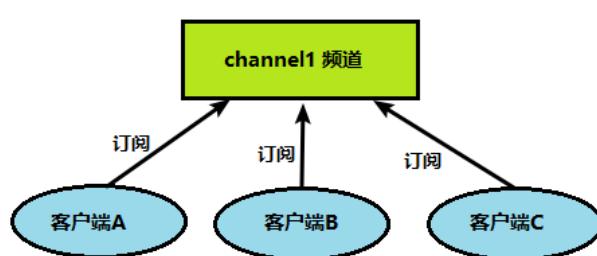
### 5.1. 什么是发布和订阅

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。

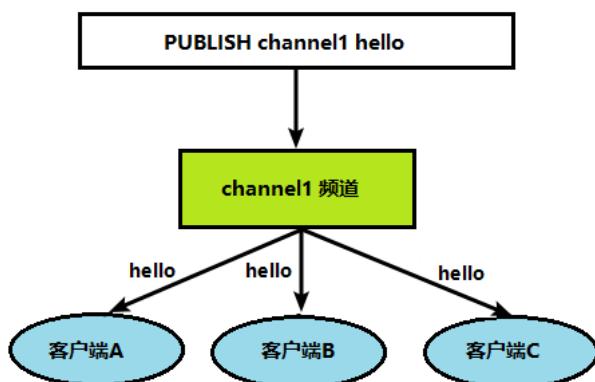
Redis 客户端可以订阅任意数量的频道。

### 5.2. Redis 的发布和订阅

1、客户端可以订阅频道如下图



2、当给这个频道发布消息后，消息就会发送给订阅的客户端



### 5.3. 发布订阅命令行实现

1、打开一个客户端订阅 channel1

```
SUBSCRIBE channel1
```

```
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 channel1 发布消息 hello

```
publish channel1 hello
```

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

返回的 1 是订阅者数量

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"
```

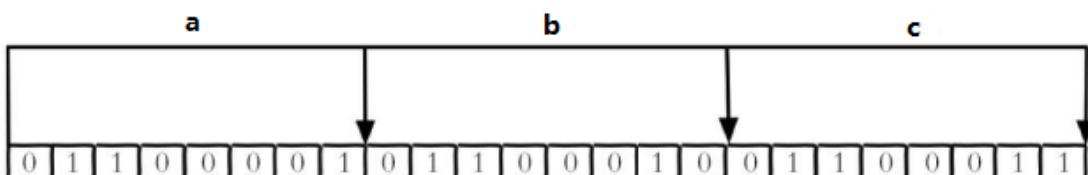
注：发布的消息没有持久化，如果在订阅的客户端收不到 hello，只能收到订阅后发布的消息

## 6. Redis 新数据类型

### 6.1. Bitmaps

#### 6.1.1. 简介

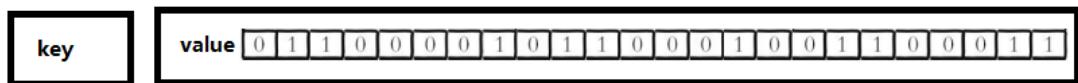
现代计算机用二进制（位）作为信息的基础单位，1个字节等于8位，例如“abc”字符串是由3个字节组成，但实际在计算机存储时将其用二进制表示，“abc”分别对应的ASCII码分别是97、98、99，对应的二进制分别是01100001、01100010和01100011，如下图



合理地使用操作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“数据类型”可以实现对位的操作：

- (1) Bitmaps 本身不是一种数据类型，实际上它就是字符串（key-value），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。



## 6.1.2. 命令

### 1、setbit

#### (1) 格式

`setbit<key><offset><value>`设置 Bitmaps 中某个偏移量的值 (0 或 1)

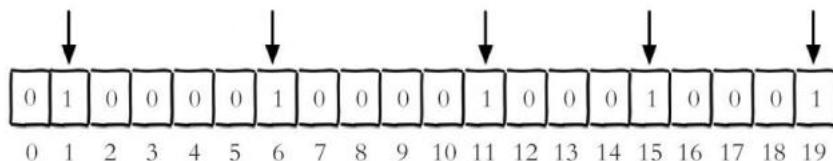
```
127.0.0.1:6379> setbit key offset value
```

\*offset:偏移量从 0 开始

#### (2) 实例

每个独立用户是否访问过网站存放在 Bitmaps 中， 将访问的用户记做 1， 没有访问的用户记做 0， 用偏移量作为用户的 id。

设置键的第 offset 个位的值 (从 0 算起)， 假设现在有 20 个用户， userid=1, 6, 11, 15, 19 的用户对网站进行了访问， 那么当前 Bitmaps 初始化结果如图



`unique:users:20201106` 代表 2020-11-06 这天的独立访问用户的 Bitmaps

```
127.0.0.1:6379> setbit unique:users:20201106 1 1
(integer) 0
127.0.0.1:6379> setbit unique:users:20201106 6 1
(integer) 0
127.0.0.1:6379> setbit unique:users:20201106 11 1
(integer) 0
127.0.0.1:6379> setbit unique:users:20201106 15 1
(integer) 0
127.0.0.1:6379> setbit unique:users:20201106 19 1
(integer) 0
```

注：

很多应用的用户 id 以一个指定数字 (例如 10000) 开头， 直接将用户 id 和 Bitmaps 的偏移量对应势必会造成一定的浪费， 通常的做法是每次做 setbit 操作时将用户 id 减去这个指定数字。

在第一次初始化 Bitmaps 时， 假如偏移量非常大， 那么整个初始化过程执行会比较慢， 可能会造成 Redis 的阻塞。

### 2、getbit

#### (1) 格式

`getbit<key><offset>`获取 Bitmaps 中某个偏移量的值

```
127.0.0.1:6379> getbit key offset
```

获取键的第 offset 位的值 (从 0 开始算)

## (2) 实例

获取 id=8 的用户是否在 2020-11-06 这天访问过，返回 0 说明没有访问过：

```
127.0.0.1:6379> getbit unique:users:20201106 8
(integer) 0
127.0.0.1:6379> getbit unique:users:20201106 1
(integer) 1
127.0.0.1:6379> getbit unique:users:20201106 100
(integer) 0
```

注：因为 100 根本不存在，所以也是返回 0

## 3、bitcount

统计**字符串**被设置为 1 的 bit 数。一般情况下，给定的整个字符串都会被进行计数，通过指定额外的 start 或 end 参数，可以让计数只在特定的位上进行。start 和 end 参数的设置，都可以使用负数值：比如 -1 表示最后一个位，而 -2 表示倒数第二个位，start、end 是指 bit 组的字节的下标数，二者皆包含。

### (1) 格式

bitcount<key>[start end] 统计字符串从 start 字节到 end 字节比特值为 1 的数量

```
127.0.0.1:6379> bitcount key [start end]
```

### (2) 实例

计算 2022-11-06 这天的独立访问用户数量

```
127.0.0.1:6379> bitcount unique:users:20201106
(integer) 5
```

start 和 end 代表起始和结束字节数，下面操作计算用户 id 在第 1 个字节到第 3 个字节之间的独立访问用户数，对应的用户 id 是 11, 15, 19。

```
127.0.0.1:6379> bitcount unique:users:20201106 1 3
(integer) 3
```

**举例： K1 【01000001 01000000 00000000 00100001】，对应【0, 1, 2, 3】**

bitcount K1 1 2 : 统计下标 1、2 字节组中 bit=1 的个数，即 01000000 00000000  
--> bitcount K1 1 2 --> 1

bitcount K1 1 3 : 统计下标 1、2 字节组中 bit=1 的个数，即 01000000 00000000  
00100001  
--> bitcount K1 1 3 --> 3

bitcount K1 0 -2 : 统计下标 0 到下标倒数第 2，字节组中 bit=1 的个数，即

```
01000001 01000000 00000000  
--> bitcount K1 0 -2      --> 3
```

注意：redis 的 setbit 设置或清除的是 bit 位置，而 bitcount 计算的是 byte 位置。

#### 4、bitop

##### (1)格式

```
bitop and(or/not/xor) <destkey> [key...]
```

```
127.0.0.1:6379> bitop operation destkey key [key ...]
```

bitop 是一个复合操作，它可以做多个 Bitmaps 的 and (交集) 、 or (并集) 、 not (非) 、 xor (异或) 操作并将结果保存在 destkey 中。

##### (2)实例

2020-11-04 日访问网站的 userid=1,2,5,9。

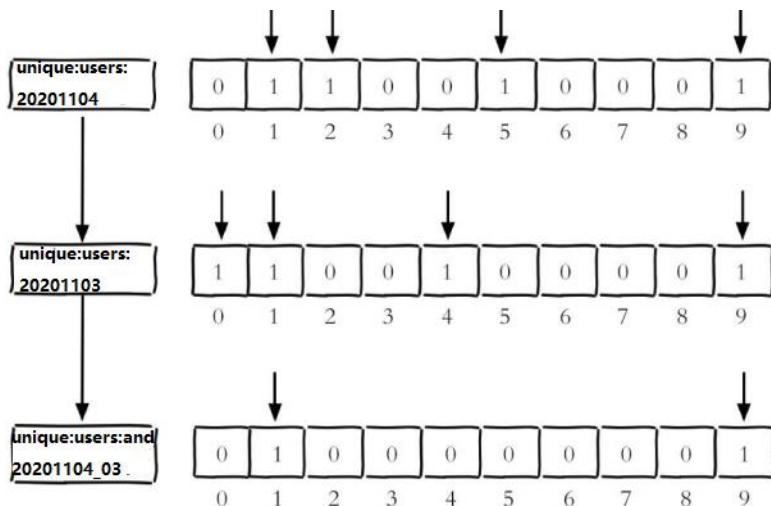
```
setbit unique:users:20201104 1 1  
setbit unique:users:20201104 2 1  
setbit unique:users:20201104 5 1  
setbit unique:users:20201104 9 1
```

2020-11-03 日访问网站的 userid=0,1,4,9。

```
setbit unique:users:20201103 0 1  
setbit unique:users:20201103 1 1  
setbit unique:users:20201103 4 1  
setbit unique:users:20201103 9 1
```

计算出两天都访问过网站的用户数量

```
bitop and unique:users:and:20201104_03  
unique:users:20201103unique:users:20201104  
127.0.0.1:6379> bitop and unique:users:and:20201104_03 unique:users:20201103 unique:users:20201104  
(integer) 2  
127.0.0.1:6379> bitcount unique:users:and:20201104_03  
(integer) 2
```



计算出任意一天都访问过网站的用户数量（例如月活跃就是类似这种），可以使用 or 求并集

```
127.0.0.1:6379> bitop or unique:users:or:20201104_03 unique:users:20201104
01104
(integer) 2
127.0.0.1:6379> bitcount unique:users:or:20201104_03
(integer) 6
```

### 6.1.3. Bitmaps 与 set 对比

假设网站有 1 亿用户，每天独立访问的用户有 5 千万，如果每天用集合类型和 Bitmaps 分别存储活跃用户可以得到表

set 和 Bitmaps 存储一天活跃用户对比			
数据类型	每个用户 id 占用空间	需要存储的用户量	全部内存量
集合类型	64 位	50000000	64 位*50000000 = 400MB
Bitmaps	1 位	100000000	1 位*100000000 = 12.5MB

很明显，这种情况下使用 Bitmaps 能节省很多的内存空间，尤其是随着时间推移节省的内存还是非常可观的

set 和 Bitmaps 存储独立用户空间对比			
数据类型	一天	一个月	一年
集合类型	400MB	12GB	144GB
Bitmaps	12.5MB	375MB	4.5GB

但 Bitmaps 并不是万金油，假如该网站每天的独立访问用户很少，例如只有 10 万（大量的僵尸用户），那么两者的对比如下表所示，很显然，这时候使用 Bitmaps 就不太合适了，因为基本上大部分位都是 0。

set 和 Bitmaps 存储一天活跃用户对比 (独立用户比较少)			
数据类型	每个 userid 占用空间	需要存储的用户量	全部内存量
集合类型	64 位	100000	64 位*100000 = 800KB
Bitmaps	1 位	1000000000	1 位*1000000000 = 12.5MB

## 6.2. HyperLogLog

### 6.2.1. 简介

在工作当中，我们经常会遇到与统计相关的功能需求，比如统计网站 PV (PageView 页面访问量)，可以使用 Redis 的 incr、incrby 轻松实现。

但像 UV (UniqueVisitor，独立访客)、独立 IP 数、搜索记录数等需要去重和计数的问题如何解决？这种求集合中不重复元素个数的问题称为基数问题。

解决基数问题有很多种方案：

- (1) 数据存储在 MySQL 表中，使用 distinct count 计算不重复个数
- (2) 使用 Redis 提供的 hash、set、bitmaps 等数据结构来处理

以上的方案结果精确，但随着数据不断增加，导致占用空间越来越大，对于非常大的数据集是不切实际的。

能否能够降低一定的精度来平衡存储空间？Redis 推出了 HyperLogLog

Redis HyperLogLog 是用来做基数统计的算法，HyperLogLog 的优点是，在输入元素的数量或者体积非常非常大时，计算基数所需的空间总是固定的、并且是很小的。

在 Redis 里面，每个 HyperLogLog 键只需要花费 12 KB 内存，就可以计算接近  $2^{64}$  个不同元素的基数。这和计算基数时，元素越多耗费内存就越多的集合形成鲜明对比。

但是，因为 HyperLogLog 只会根据输入元素来计算基数，而不会储存输入元素本身，所以 HyperLogLog 不能像集合那样，返回输入的各个元素。

什么是基数？

比如数据集 {1, 3, 5, 7, 5, 7, 8}，那么这个数据集的基数集为 {1, 3, 5, 7, 8}，基数(不重复元素)为 5。基数估计就是在误差可接受的范围内，快速计算基数。

### 6.2.2. 命令

1、pfadd

### (1) 格式

pfadd <key>< element> [element ...] 添加指定元素到 HyperLogLog 中

```
127.0.0.1:6379> pfadd key element [element ...]
```

### (2) 实例

```
127.0.0.1:6379> pfadd hll1 "redis"
(integer) 1
127.0.0.1:6379> pfadd hll1 "mysql"
(integer) 1
127.0.0.1:6379> pfadd hll1 "redis"
(integer) 0
127.0.0.1:6379>
```

将所有元素添加到指定 HyperLogLog 数据结构中。如果执行命令后 HLL 估计的近似基数发生变化，则返回 1，否则返回 0。

## 2、pfcount

### (1) 格式

pfcount<key> [key ...] 计算 HLL 的近似基数，可以计算多个 HLL，比如用 HLL 存储每天的 UV，计算一周的 UV 可以使用 7 天的 UV 合并计算即可

```
127.0.0.1:6379> pfcount key [key ...]
```

### (2) 实例

```
127.0.0.1:6379> pfadd hll1 "redis"
(integer) 1
127.0.0.1:6379> pfadd hll1 "mysql"
(integer) 1
127.0.0.1:6379> pfadd hll1 "redis"
(integer) 0
127.0.0.1:6379> pfcount hll1
(integer) 2
127.0.0.1:6379> pfadd hll2 "redis"
(integer) 1
127.0.0.1:6379> pfadd hll2 "mongodb"
(integer) 1
127.0.0.1:6379> pfcount hll1 hll2
(integer) 3
127.0.0.1:6379>
```

## 3、pfmerge

### (1) 格式

pfmerge<destkey><sourcekey> [sourcekey ...] 将一个或多个 HLL 合并后的结果存储在另一个 HLL 中，比如每月活跃用户可以使用每天的活跃用户来合并计算可得

```
127.0.0.1:6379> pfmerge destkey sourcekey [sourcekey ...]
```

### (2) 实例

```
127.0.0.1:6379> pfcount hll1 hll2
(integer) 3
127.0.0.1:6379> pfmerge hll3 hll1 hll2
OK
127.0.0.1:6379> pfcount hll3
(integer) 3
127.0.0.1:6379> █
```

## 6.3. Geospatial

### 6.3.1. 简介

Redis 3.2 中增加了对 GEO 类型的支持。GEO，Geographic，地理信息的缩写。该类型，就是元素的 2 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。

### 6.3.2. 命令

#### 1、geoadd

##### (1) 格式

geoadd<key><longitude><latitude><member> [longitude latitude member...] 添加地理位置 (经度, 纬度, 名称)

```
127.0.0.1:6379> geoadd █ key longitude latitude member [longitude latitude me
```

##### (2) 实例

```
geoadd china:city 121.47 31.23 shanghai
geoadd china:city 106.50 29.53 chongqing 114.05 22.52 shenzhen 116.38 39.90
beijing
```

```
127.0.0.1:6379> geoadd china:city 121.47 31.23 shanghai
(integer) 1
127.0.0.1:6379> geoadd china:city 106.50 29.53 chongqing 114.05 22.52 shenzhen 116.38 39.90 beijing
(integer) 3
127.0.0.1:6379> █
```

两极无法直接添加，一般会下载城市数据，直接通过 Java 程序一次性导入。

有效的经度从 -180 度到 180 度。有效的纬度从 -85.05112878 度到 85.05112878 度。

当坐标位置超出指定范围时，该命令将会返回一个错误。

已经添加的数据，是无法再次往里面添加的。

#### 2、geopos

## (1) 格式

geopos <key><member> [member...] 获得指定地区的坐标值

```
127.0.0.1:6379> geopos key member [member ...]
```

## (2) 实例

```
127.0.0.1:6379> geopos china:city shanghai
```

```
1) 1) "121.47000163793563843"
```

```
2) "31.22999903975783553"
```

## 3、geodist

## (1) 格式

geodist<key><member1><member2> [m|km|ft|mi] 获得两个位置之间的直线距离

```
127.0.0.1:6379> geodist key member1 member2 [m|km|ft|mi]
```

## (2) 实例

获取两个位置之间的直线距离

```
127.0.0.1:6379> geodist china:city beijing shanghai km  
"1087.4816"
```

单位：

m 表示单位为米[默认值]。

km 表示单位为千米。

mi 表示单位为英里。

ft 表示单位为英尺。

如果用户没有显式地指定单位参数，那么 GEODIST 默认使用米作为单位

## 4、georadius

## (1) 格式

georadius<key>< longitude><latitude>radius m|km|ft|mi 以给定的经纬度为中心，找出某一半径内的元素

```
127.0.0.1:6379> georadius key longitude latitude radius m|km|ft|mi
```

经度 纬度 距离 单位

## (2) 实例

```
127.0.0.1:6379> georadius china:city 110 30 1000 km
1) "chongqing"
2) "shengzhen"
```

## 7. Redis\_Jedis\_ 测试

### 7.1. Jedis 所需要的 jar 包

```
<dependency>
<groupId>redis.clients</groupId>
<artifactId>jedis</artifactId>
<version>3.2.0</version>
</dependency>
```

### 7.2. 连接 Redis 注意事项

禁用 Linux 的防火墙：Linux(CentOS7)里执行命令

**systemctl stop/disable firewalld.service**

redis.conf 中注释掉 bind 127.0.0.1 ,然后 protected-mode no

### 7.3. Jedis 常用操作

#### 7.3.1. 创建动态的工程

#### 7.3.2. 创建测试程序

```
package com.atguigu.jedis;

import redis.clients.jedis.Jedis;

public class Demo01 {

    public static void main(String[] args) {
        Jedis jedis = new Jedis("192.168.137.3",6379);
        String pong = jedis.ping();
        System.out.println("连接成功: "+pong);
        jedis.close();
    }
}
```

```
}
```

## 7.4. 测试相关数据类型

### 7.4.1. Redis-API: Key

```
jedis.set("k1", "v1");
jedis.set("k2", "v2");
jedis.set("k3", "v3");
Set<String> keys = jedis.keys("*");
System.out.println(keys.size());
for (String key : keys) {
    System.out.println(key);
}
System.out.println(jedis.exists("k1"));
System.out.println(jedis.ttl("k1"));
System.out.println(jedis.get("k1"));
```

### 7.4.2. Redis-API: String

```
jedis.mset("str1","v1","str2","v2","str3","v3");
System.out.println(jedis.mget("str1","str2","str3"));
```

### 7.4.3. Redis-API: List

```
List<String> list = jedis.lrange("mylist",0,-1);
for (String element : list) {
    System.out.println(element);
}
```

### 7.4.4. Redis-API: set

```
jedis.sadd("orders", "order01");
```

```
jedis.sadd("orders", "order02");
jedis.sadd("orders", "order03");
jedis.sadd("orders", "order04");
Set<String> smembers = jedis.smembers("orders");
for (String order : smembers) {
    System.out.println(order);
}
jedis.srem("orders", "order02");
```

#### 7.4.5.     Jedis-API:     hash

```
jedis.hset("hash1", "userName", "lisi");
System.out.println(jedis.hget("hash1", "userName"));
Map<String, String> map = new HashMap<String, String>();
map.put("telephone", "13810169999");
map.put("address", "atguigu");
map.put("email", "abc@163.com");
jedis.hmset("hash2", map);
List<String> result = jedis.hmget("hash2", "telephone", "email");
for (String element : result) {
    System.out.println(element);
}
```

#### 7.4.6.     Jedis-API:     zset

```
jedis.zadd("zset01", 100d, "z3");
jedis.zadd("zset01", 90d, "l4");
jedis.zadd("zset01", 80d, "w5");
jedis.zadd("zset01", 70d, "z6");

Set<String> zrange = jedis.zrange("zset01", 0, -1);
```

```
for (String e : zrange) {  
    System.out.println(e);  
}
```

## 8. Redis\_Jedis\_实例

### 8.1. 完成一个手机验证码功能

要求：

- 1、输入手机号，点击发送后随机生成 6 位数字码，2 分钟有效
- 2、输入验证码，点击验证，返回成功或失败
- 3、每个手机号每天只能输入 3 次



## 9. Redis 与 Spring Boot 整合

Spring Boot 整合 Redis 非常简单，只需要按如下步骤整合即可

### 9.1. 整合步骤

- 1、在 pom.xml 文件中引入 redis 相关依赖

```
<!-- redis -->  
<dependency>  
<groupId>org.springframework.boot</groupId>  
<artifactId>spring-boot-starter-data-redis</artifactId>  
</dependency>  
  
<!-- spring2.X 集成redis 所需 common-pool2-->  
<dependency>  
<groupId>org.apache.commons</groupId>
```

```
<artifactId>commons-pool2</artifactId>
<version>2.6.0</version>
</dependency>
```

## 2、application.properties 配置 redis 配置

```
#Redis 服务器地址
spring.redis.host=192.168.140.136
#Redis 服务器连接端口
spring.redis.port=6379
#Redis 数据库索引（默认为0）
spring.redis.database= 0
#连接超时时间（毫秒）
spring.redis.timeout=1800000
#连接池最大连接数（使用负值表示没有限制）
spring.redis.lettuce.pool.max-active=20
#最大阻塞等待时间（负数表示没限制）
spring.redis.lettuce.pool.max-wait=-1
#连接池中的最大空闲连接
spring.redis.lettuce.pool.max-idle=5
#连接池中的最小空闲连接
spring.redis.lettuce.pool.min-idle=0
```

## 3、添加 redis 配置类

```
@EnableCaching
@Configuration
public class RedisConfig extends CachingConfigurerSupport {

    @Bean
    public RedisTemplate<String, Object> redisTemplate(RedisConnectionFactory
factory) {
        RedisTemplate<String, Object> template = new RedisTemplate<>();
        RedisSerializer<String> redisSerializer = new StringRedisSerializer();
        Jackson2JsonRedisSerializer jackson2JsonRedisSerializer = new
Jackson2JsonRedisSerializer(Object.class);
        ObjectMapper om = new ObjectMapper();
        om.setVisibility(PropertyAccessor.ALL, JsonAutoDetect.Visibility.ANY);
        om.enableDefaultTyping(ObjectMapper.DefaultTyping.NON_FINAL);
        jackson2JsonRedisSerializer.setObjectMapper(om);
        template.setConnectionFactory(factory);
        //key 序列化方式
        template.setKeySerializer(redisSerializer);
        //value 序列化
        template.setValueSerializer(jackson2JsonRedisSerializer);
        //value hashmap 序列化
        template.setHashValueSerializer(jackson2JsonRedisSerializer);
        return template;
    }
}
```

```
}

@Bean
public CacheManager cacheManager(RedisConnectionFactory factory) {
    RedisSerializer<String> redisSerializer = new StringRedisSerializer();
    Jackson2JsonRedisSerializer jackson2JsonRedisSerializer = new
Jackson2JsonRedisSerializer(Object.class);
//解决查询缓存转换异常的问题
    ObjectMapper om = new ObjectMapper();
    om.setVisibility(PropertyAccessor.ALL, JsonAutoDetect.Visibility.ANY);
    om.enableDefaultTyping(ObjectMapper.DefaultTyping.NON_FINAL);
    jackson2JsonRedisSerializer.setObjectMapper(om);
// 配置序列化（解决乱码的问题），过期时间 600 秒
    RedisCacheConfiguration config =
RedisCacheConfiguration.defaultCacheConfig()
        .entryTtl(Duration.ofSeconds(600))
        .serializeKeysWith(RedisSerializationContext.SerializationPair.
fromSerializer(redisSerializer))
        .serializeValuesWith(RedisSerializationContext.SerializationPai
r.fromSerializer(jackson2JsonRedisSerializer))
        .disableCachingNullValues();
    RedisCacheManager cacheManager = RedisCacheManager.builder(factory)
        .cacheDefaults(config)
        .build();
    return cacheManager;
}
}
```

4、测试一下  
RedisTestController 中添加测试方法

```
@RestController
@RequestMapping("/redisTest")
public class RedisTestController {
    @Autowired
    private RedisTemplate redisTemplate;

    @GetMapping
    public String testRedis() {
        //设置值到redis
        redisTemplate.opsForValue().set("name", "lucy");
        //从 redis 获取值
        String name = (String) redisTemplate.opsForValue().get("name");
        return name;
    }
}
```

## 10. Redis\_事务\_锁机制\_秒杀

### 10.1. Redis 的事务定义

- All the commands in a transaction are serialized and executed sequentially. It can never happen that a request issued by another client is served **in the middle** of the execution of a Redis transaction. This guarantees that the commands are executed as a single isolated operation.

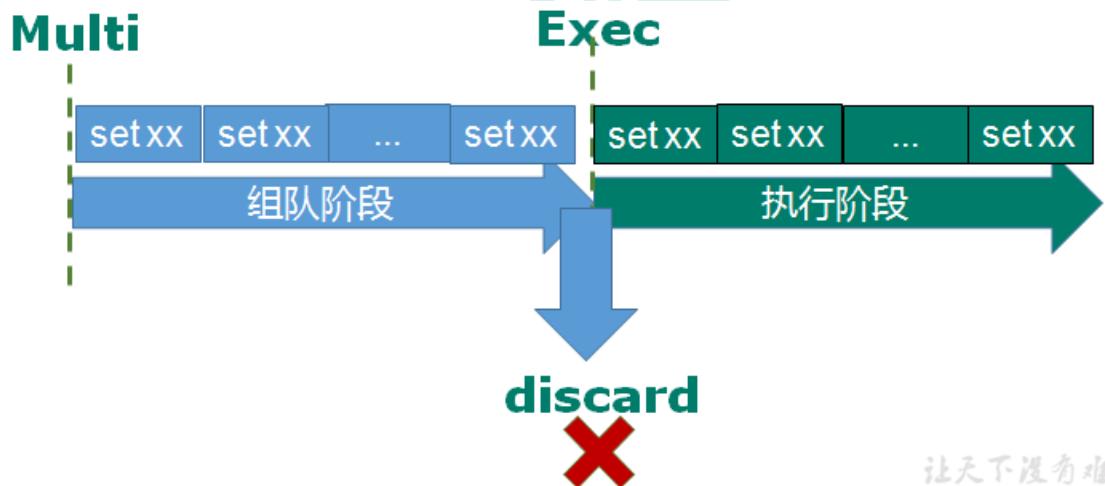
Redis 事务是一个单独的隔离操作：事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。

Redis 事务的主要作用就是**串联多个命令**防止别的命令插队。

### 10.2. Multi、Exec、discard

从输入 Multi 命令开始，输入的命令都会依次进入命令队列中，但不会执行，直到输入 Exec 后，Redis 会将之前的命令队列中的命令依次执行。

组队的过程中可以通过 discard 来放弃组队。



案例：

```
127.0.0.1:6379> multi
OK
127.0.0.1:6379> set k2 v2
QUEUED
127.0.0.1:6379> set k3 v3
QUEUED
127.0.0.1:6379> exec
1) OK
2) OK
```

组队成功，提交成功

```
127.0.0.1:6379> multi
OK
127.0.0.1:6379> set m1 v1
QUEUED
127.0.0.1:6379> set m2
(error) ERR wrong number of arguments for 'set' command
127.0.0.1:6379> set m3 v3
QUEUED
127.0.0.1:6379> exec
(error) EXECABORT Transaction discarded because of previous errors.
```

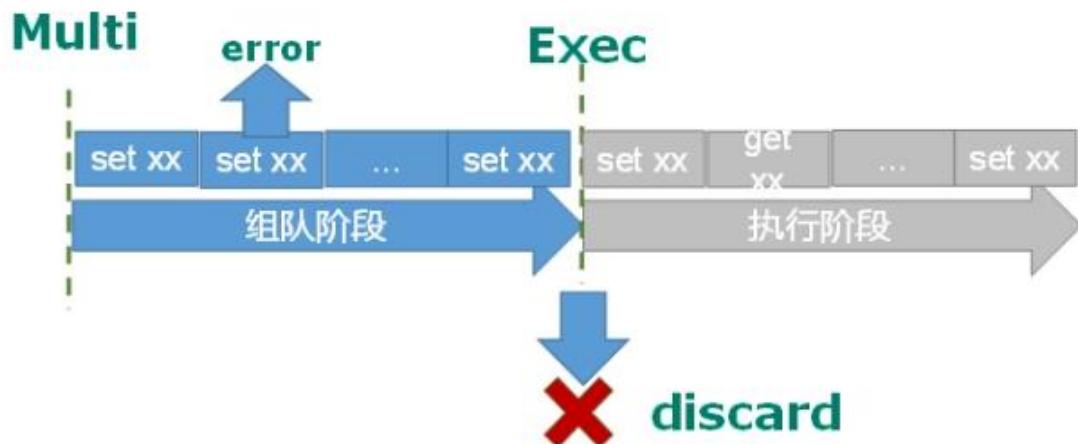
组队阶段报错，提交失败

```
127.0.0.1:6379> multi
OK
127.0.0.1:6379> set m1 v1
QUEUED
127.0.0.1:6379> incr m1
QUEUED
127.0.0.1:6379> set m2 v2
QUEUED
127.0.0.1:6379> exec
1) OK
2) (error) ERR value is not an integer or out of range
3) OK
```

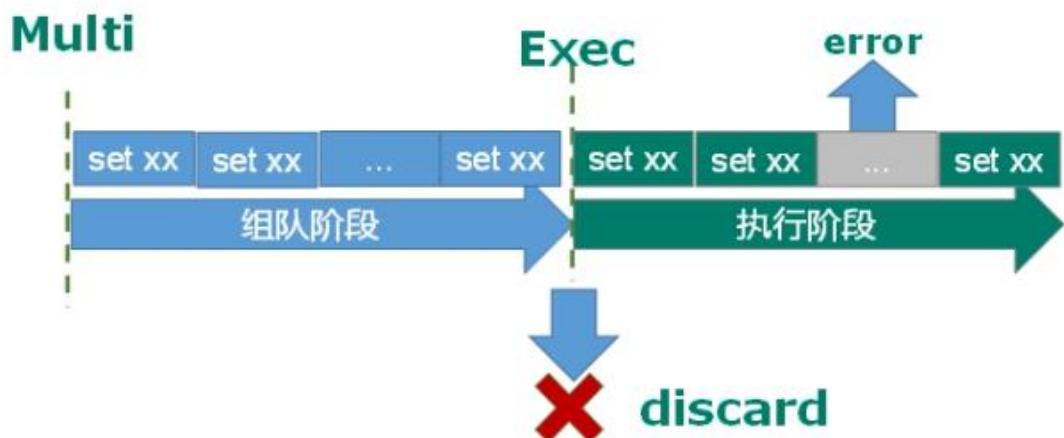
组队成功，提交有成功有失败情况

### 10.3. 事务的错误处理

组队中某个命令出现了报告错误，执行时整个的所有队列都会被取消。



如果执行阶段某个命令报出了错误，则只有报错的命令不会被执行，而其他的命令都会执行，不会回滚。



## 10.4. 为什么要做成事务

想想一个场景：有很多人有你的账户，同时去参加双十一抢购

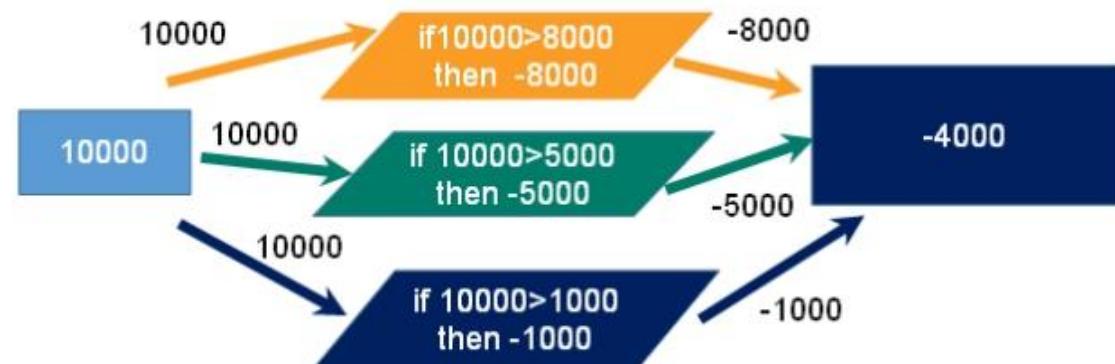
## 10.5. 事务冲突的问题

### 10.5.1. 例子

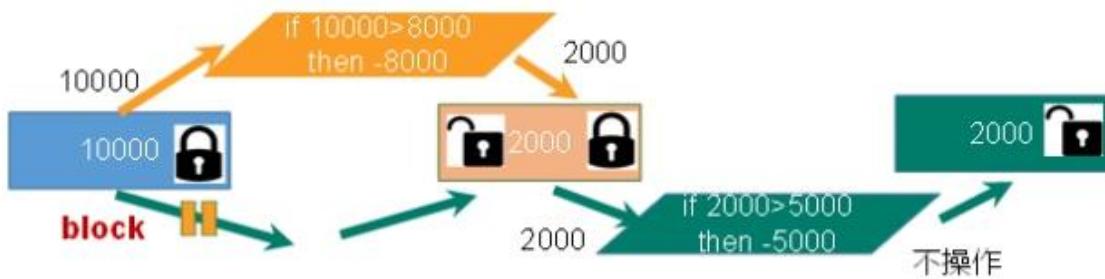
一个请求想给金额减 8000

一个请求想给金额减 5000

一个请求想给金额减 1000

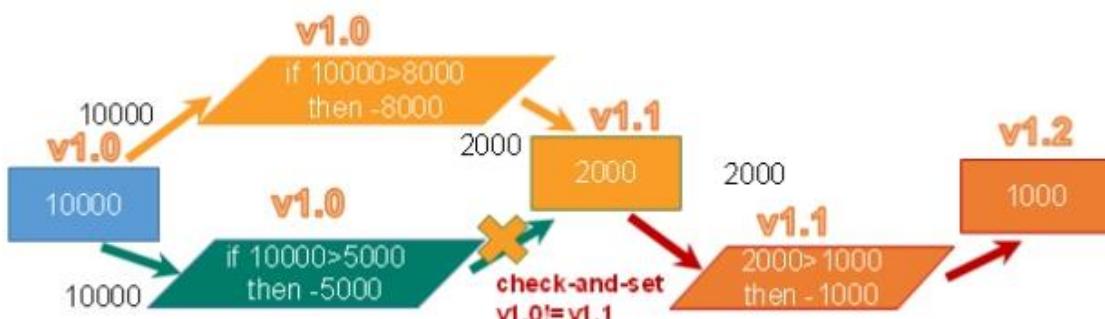


### 10.5.2. 悲观锁



**悲观锁(Pessimistic Lock)**, 顾名思义, 就是很悲观, 每次去拿数据的时候都认为别人会修改, 所以每次在拿数据的时候都会上锁, 这样别人想拿这个数据就会 block 直到它拿到锁。传统的关系型数据库里边就用到了很多这种锁机制, 比如**行锁**, **表锁**等, **读锁**, **写锁**等, 都是在做操作之前先上锁。

### 10.5.3. 乐观锁



**乐观锁(Optimistic Lock)**, 顾名思义, 就是很乐观, 每次去拿数据的时候都认为别人不会修改, 所以不会上锁, 但是在更新的时候会判断一下在此期间别人有没有去更新这个数据, 可以使用版本号等机制。**乐观锁适用于多读的应用类型, 这样可以提高吞吐量**。Redis 就是利用这种 check-and-set 机制实现事务的。

### 10.5.4. WATCH key [key ...]

在执行 `multi` 之前, 先执行 `watch key1 [key2]`, 可以监视一个(或多个) key , 如果在事务执行之前这个(或这些) key 被其他命令所改动, 那么事务将被打断。

```
127.0.0.1:6379> WATCH balance
OK
127.0.0.1:6379> MULTI
OK
127.0.0.1:6379> DECRBY balance 10
QUEUED
127.0.0.1:6379> INCRBY debt 10
QUEUED
127.0.0.1:6379> EXEC
1) (integer) 60
2) (integer) 40
.
```

### 10.5.5. 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 命令先被执行了的话，那么就不再需要再执行 UNWATCH 了。

<http://doc.redisfans.com/transaction/exec.html>

## 10.6. Redis 事务三特性

- 单独的隔离操作
  - 事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。
- 没有隔离级别的概念
  - 队列中的命令没有提交之前都不会实际被执行，因为事务提交前任何指令都不会被实际执行
- 不保证原子性
  - 事务中如果有一条命令执行失败，其后的命令仍然会被执行，没有回滚

## 11. Redis\_事务\_秒杀案例

### 11.1. 解决计数器和人员记录的事务操作



### 11.2. Redis 事务--秒杀并发模拟

使用工具 ab 模拟测试

CentOS6 默认安装

CentOS7 需要手动安装

#### 11.2.1. 联网: yum install httpd-tools

#### 11.2.2. 无网络

(1) 进入 cd /run/media/root/CentOS 7 x86\_64/Packages (路径跟 centos6 不同)

(2) 顺序安装

apr-1.4.8-3.el7.x86\_64.rpm

apr-util-1.5.2-6.el7.x86\_64.rpm

httpd-tools-2.4.6-67.el7.centos.x86\_64.rpm

#### 11.2.3. 测试及结果

##### 11.2.3.1. 通过 ab 测试

vim postfile 模拟表单提交参数,以&符号结尾;存放当前目录。

内容: prodid=0101&

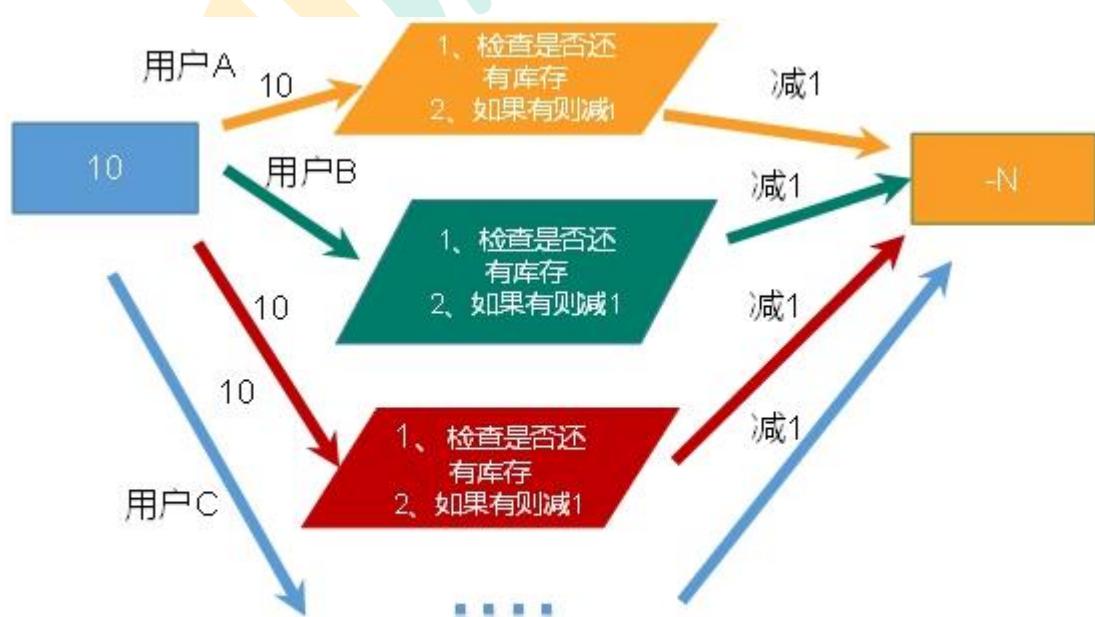
```
ab -n 2000 -c 200 -k -p ~/postfile -T application/x-www-form-urlencoded
```

```
http://192.168.2.115:8081/Seckill/doseckill
```

### 11.2.3.2. 超卖

秒杀成功! ! ! ! 秒杀成功! ! ! ! 已秒光! ! 秒杀成功! ! ! ! 已秒光! ! 已秒光! ! 已秒光! ! 已秒光! !	127.0.0.1:6379> del sk:0101:usr (integer) 1 127.0.0.1:6379> set sk:0101:qt 10 OK 127.0.0.1:6379> get sk:0101:qt "-1"
---	---

### 11.3. 超卖问题



## 11.4. 利用乐观锁淘汰用户，解决超卖问题。



```
//增加乐观锁
jedis.watch(qtkey);

//3.判断库存
String qtkeystr = jedis.get(qtkey);
if(qtkeystr==null ||
"".equals(qtkeystr.trim())) {
System.out.println("未初始化库存");
jedis.close();
return false ;
}

int qt = Integer.parseInt(qtkeystr);
if(qt<=0) {
System.err.println("已经秒光");
jedis.close();
return false;
}
```

```
127.0.0.1:6379> del sk:0101:usr
(integer) 1
127.0.0.1:6379> set sk:0101:qt 10
OK
127.0.0.1:6379> smembers sk:0101:usr
1) "129"
2) "2368"
3) "7153"
4) "10909"
5) "17259"
6) "19148"
7) "21496"
8) "23318"
9) "26097"
10) "26642"
127.0.0.1:6379> get sk:0101:qt
"0"
```

```
//增加事务  
Transaction multi = jedis.multi();  
  
//4.减少库存  
//jedis.decr(qtkey);  
multi.decr(qtkey);  
  
//5.加入  
//jedis.sadd(usrkey, uid);  
multi.sadd(usrkey, uid);  
  
//执行事务  
List<Object> list = multi.exec();  
  
//判断事务提交是否失败  
if(list==null || list.size()==0) {  
    System.out.println("秒杀失败");  
    jedis.close();  
    return false;  
}  
System.err.println("秒杀成功");  
jedis.close();
```

## 11.5. 继续增加并发测试

### 11.5.1. 连接有限制

```
ab -n 2000 -c 200 -k -p postfile -T 'application/x-www-form-urlencoded'  
http://192.168.140.1:8080/seckill/doseckill
```

```
[root@zhangyu ~]# ab -n 2000 -c 200 -p postfile -T 'application/x-www-form-urlencoded' http://192.168.137.1:8080/seckill/doseckill
This is ApacheBench, Version 2.3 <Revision: 1438300 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking 192.168.137.1 (be patient)
apr_socket_recv: Connection refused (111)
Total of 1 requests completed
```

增加-r 参数， -r Don't exit on socket receive errors.

```
ab -n 2000 -c 100 -r -p postfile -T 'application/x-www-form-urlencoded'
http://192.168.140.1:8080/seckill/doseckill
```

### 11.5.2. 已经秒光，可是还有库存

```
ab -n 2000 -c 100 -p postfile -T 'application/x-www-form-urlencoded'
http://192.168.137.1:8080/seckill/doseckill
```

已经秒光，可是还有库存。原因，就是**乐观锁导致很多请求都失败**。先点的没秒到，后点的可能秒到了。

```
127.0.0.1:6379> del sk:0101:usr
(integer) 1
127.0.0.1:6379> set sk:0101:qt 200
OK
127.0.0.1:6379> get sk:0101:qt
"140"
```

### 11.5.3. 连接超时，通过连接池解决

```
core.StandardWrapperValve invoke
[] in context with path [/seckill] threw exception
onException: java.net.SocketTimeoutException: connect timed out
connect\(URLConnection.java:207\)
```

### 11.5.4. 连接池

节省每次连接 redis 服务带来的消耗，把连接好的实例反复利用。

通过参数管理连接的行为

代码见项目中

- 链接池参数
  - MaxTotal：控制一个 pool 可分配多少个 jedis 实例，通过 pool.getResource() 来获取；如果赋值为-1，则表示不限制；如果 pool 已经分配了 MaxTotal 个 jedis 实例，则此时 pool 的状态为 exhausted。
  - maxIdle：控制一个 pool 最多有多少个状态为 idle(空闲)的 jedis 实例；
  - MaxWaitMillis：表示当 borrow 一个 jedis 实例时，最大的等待毫秒数，如果超过等待时间，则直接抛 JedisConnectionException；

- testOnBorrow: 获得一个 jedis 实例的时候是否检查连接可用性 (ping()) ; 如果为 true, 则得到的 jedis 实例均是可用的;

## 11.6. 解决库存遗留问题

### 11.6.1. LUA 脚本



Lua 是一个小巧的[脚本语言](#), Lua 脚本可以很容易的被 C/C++ 代码调用, 也可以反过来调用 C/C++ 的函数, Lua 并没有提供强大的库, 一个完整的 Lua 解释器不过 200k, 所以 Lua 不适合作为开发独立应用程序的语言, 而是作为[嵌入式脚本语言](#)。

很多应用程序、游戏使用 LUA 作为自己的嵌入式脚本语言, 以此来实现可配置性、可扩展性。

这其中包括魔兽争霸地图、魔兽世界、博德之门、愤怒的小鸟等众多游戏插件或外挂。

<https://www.w3cschool.cn/lua/>

### 11.6.2. LUA 脚本在 Redis 中的优势

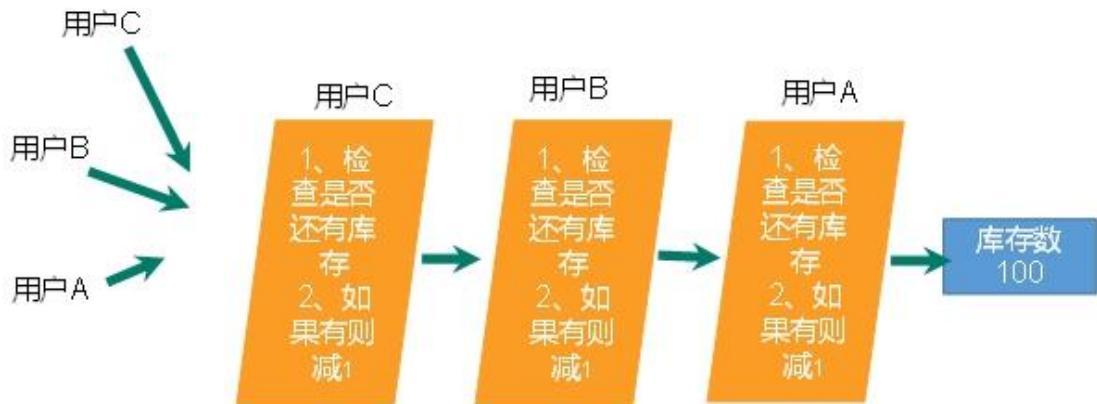
将复杂的或者多步的 redis 操作, 写为一个脚本, 一次提交给 redis 执行, 减少反复连接 redis 的次数。提升性能。

LUA 脚本是类似 redis 事务, 有一定的原子性, [不会被其他命令插队](#), 可以完成一些 redis 事务性的操作。

但是注意 redis 的 lua 脚本功能, 只有在 Redis 2.6 以上的版本才可以使用。

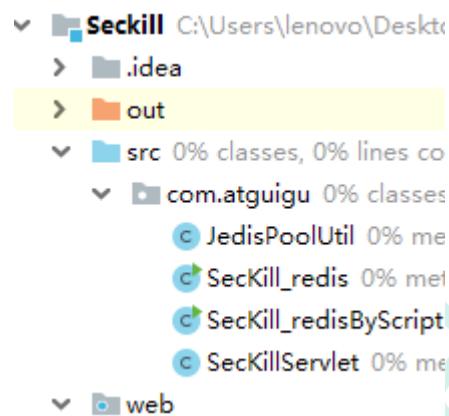
利用 lua 脚本淘汰用户, 解决超卖问题。

redis 2.6 版本以后, 通过 lua 脚本解决[争抢问题](#), 实际上是 [redis 利用其单线程的特性, 用任务队列的方式解决多任务并发问题](#)。



## 11.7. Redis\_事务\_秒杀案例\_代码

### 11.7.1. 项目结构



### 11.7.2. 第一版：简单版

老师点 10 次，正常秒杀

同学一起点试一试，秒杀也是正常的。这是因为还达不到并发的效果。

使用工具 ab 模拟并发测试，会出现超卖情况。查看库存会出现负数。

### 11.7.3. 第二版：加事务-乐观锁(解决超卖), 但出现遗留

库存和连接超时

### 11.7.4. 第三版：连接池解决超时问题

### 11.7.5. 第四版：解决库存依赖问题，LUA 脚本

```
local userid=KEYS[1];
local prodid=KEYS[2];
local qtkey="sk:..prodid..:qt";
local usersKey="sk:..prodid.:usr";
local userExists=redis.call("sismember",usersKey,userid);
if tonumber(userExists)==1 then
    return 2;
end
local num= redis.call("get" ,qtkey);
if tonumber(num)<=0 then
    return 0;
else
    redis.call("decr",qtkey);
    redis.call("sadd",usersKey,userid);
end
return 1;
```

## 12. Redis 持久化之 RDB

### 12.1. 总体介绍

官网介绍：<http://www.redis.io>

## Redis Persistence

Redis provides a different range of persistence options:

- The RDB persistence performs point-in-time snapshots of your dataset at specified intervals.
- The AOF persistence logs every write operation received by the server, that will be played again at server startup, reconstructing the original dataset. Commands are logged using the same format as the Redis protocol itself, in an append-only fashion. Redis is able to rewrite the log in the background when it gets too big.
- If you wish, you can disable persistence completely, if you want your data to just exist as long as the server is running.
- It is possible to combine both AOF and RDB in the same instance. Notice that, in this case, when Redis restarts the AOF file will be used to reconstruct the original dataset since it is guaranteed to be the most complete.

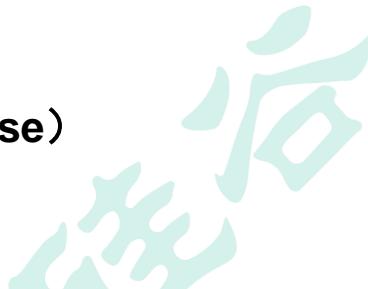
The most important thing to understand is the different trade-offs between the RDB and AOF persistence. Let's start with RDB:

Redis 提供了 2 个不同形式的持久化方式。

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

### 12.2. RDB (Redis DataBase)

#### 12.2.1. 官网介绍



##### RDB advantages

- RDB is a very compact single-file point-in-time representation of your Redis data. RDB files are perfect for backups. For instance you may want to archive your RDB files every hour for the latest 24 hours, and to save an RDB snapshot every day for 30 days. This allows you to easily restore different versions of the data set in case of disasters.
- RDB is very good for disaster recovery, being a single compact file that can be transferred to far data centers, or onto Amazon S3 (possibly encrypted).
- RDB maximizes Redis performances since the only work the Redis parent process needs to do in order to persist is forking a child that will do all the rest. The parent instance will never perform disk I/O or alike.
- RDB allows faster restarts with big datasets compared to AOF.

##### RDB disadvantages

- RDB is NOT good if you need to minimize the chance of data loss in case Redis stops working (for example after a power outage). You can configure different *save points* where an RDB is produced (for instance after at least five minutes and 100 writes against the data set, but you can have multiple save points). However you'll usually create an RDB snapshot every five minutes or more, so in case of Redis stopping working without a correct shutdown for any reason you should be prepared to lose the latest minutes of data.
- RDB needs to fork() often in order to persist on disk using a child process. Fork() can be time consuming if the dataset is big, and may result in Redis to stop serving clients for some millisecond or even for one second if the dataset is very big and the CPU performance not great. AOF also needs to fork() but you can tune how often you want to rewrite your logs without any trade-off on durability.

#### 12.2.2. 是什么

在指定的时间间隔内将内存中的数据集快照写入磁盘，也就是行话讲的 Snapshot 快照，它恢复时是将快照文件直接读到内存里

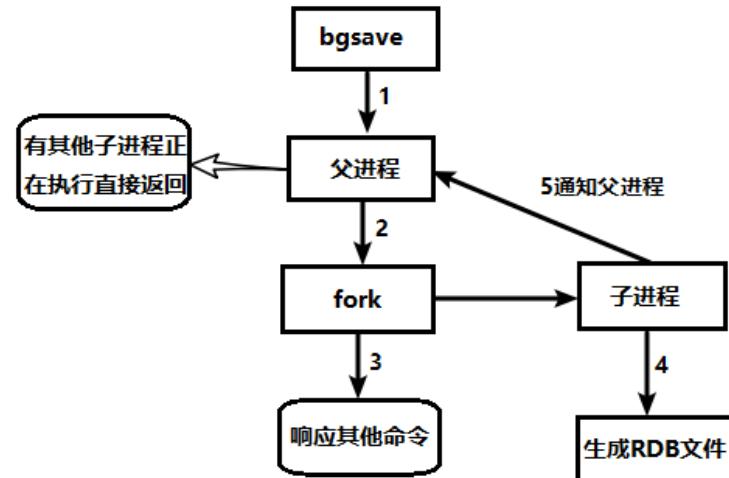
### 12.2.3. 备份是如何执行的

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。**

### 12.2.4. Fork

- Fork 的作用是复制一个与当前进程一样的进程。新进程的所有数据（变量、环境变量、程序计数器等）数值都和原进程一致，但是是一个全新的进程，并作为原进程的子进程
- 在 Linux 程序中，fork()会产生一个和父进程完全相同的子进程，但子进程在此后多会 exec 系统调用，出于效率考虑，Linux 中引入了“**写时复制技术**”
- 一般情况父进程和子进程会共用同一段物理内存，只有进程空间的各段的内容要发生变化时，才会将父进程的内容复制一份给子进程。

### 12.2.5. RDB 持久化流程



### 12.2.6. dump. rdb 文件

在 `redis.conf` 中配置文件名称，默认为 `dump.rdb`

```
418 # Sanitize dump payload no  
419  
420 # The filename where to dump the DB  
421 dbfilename dump.rdb  
422
```

## 12.2.7. 配置位置

rdb 文件的保存路径，也可以修改。默认为 Redis 启动时命令行所在的目录下

```
dir "/myredis/"
```

```
432  
433 # The working directory.  
434 #  
435 # The DB will be written inside this directory, with the filename specified  
436 # above using the 'dbfilename' configuration directive.  
437 #  
438 # The Append Only File will also be created inside this directory.  
439 #  
440 #  
441 # Note that you must specify a directory here, not a file name.  
442 #  
443 # Note that you must specify a directory here, not a file name.  
444 dir ./  
445
```

## 12.2.8. 如何触发 RDB 快照；保持策略

### 12.2.8.1. 配置文件中默认的快照配置

```
363 #  
364 # Unless specified otherwise, by default Redis will save the DB:  
365 # * After 3600 seconds (an hour) if at least 1 key changed  
366 # * After 300 seconds (5 minutes) if at least 100 keys changed  
367 # * After 60 seconds if at least 10000 keys changed  
368 #  
369 # You can set these explicitly by uncommenting the three following lines.  
370 #  
371 save 3600 1  
372 save 30 10  
373 save 60 10000  
374
```

### 12.2.8.2. 命令 save VS bgsave

save : save 时只管保存，其它不管，全部阻塞。手动保存。不建议。

**bgsave:** 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。

可以通过 lastsave 命令获取最后一次成功执行快照的时间

### 12.2.8.3. flushall 命令

执行 flushall 命令，也会产生 dump.rdb 文件，但里面是空的，无意义

### 12.2.8.4. #####SNAPSHOTTING 快照####

#### 12.2.8.5. Save

格式： save 秒钟 写操作次数

RDB 是整个内存的压缩过的 Snapshot，RDB 的数据结构，可以配置复合的快照触发条件，

**默认是 1 分钟内改了 1 万次，或 5 分钟内改了 10 次，或 15 分钟内改了 1 次。**

禁用

不设置 save 指令，或者给 save 传入空字符串

#### 12.2.8.6. stop-writes-on-bgsave-error

```
383 #
384 # However if you have setup your proper monitoring of the Redis server
385 # and persistence, you may want to disable this feature so that Redis will
386 # continue to work as usual even if there are problems with disk,
387 # permissions, and so forth.
388 stop-writes-on-bgsave-error yes
389
```

当 Redis 无法写入磁盘的话，直接关掉 Redis 的写操作。推荐 yes.

#### 12.2.8.7. rdbcompression 压缩文件

```
389
390 # Compress string objects using LZF when dump .rdb databases?
391 # By default compression is enabled as it's almost always a win.
392 # If you want to save some CPU in the saving child set it to 'no' but
393 # the dataset will likely be bigger if you have compressible values or keys.
394 rdbcompression yes
395
```

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。推荐 yes.

### 12.2.8.8. rdbchecksum 检查完整性

```
400 #
401 # RDB files created with checksum disabled have a checksum of zero that will
402 # tell the loading code to skip the check.
403 rdbchecksum yes
404
```

在存储快照后，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

推荐 yes.

### 12.2.8.9. rdb 的备份

先通过 config get dir 查询 rdb 文件的目录

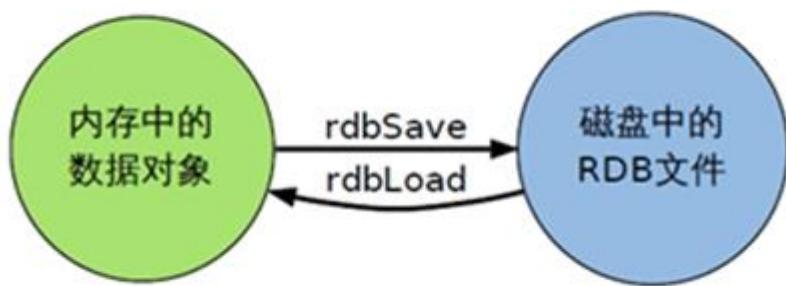
将\*.rdb 的文件拷贝到别的地方

rdb 的恢复

- ◆ 关闭 Redis
- ◆ 先把备份的文件拷贝到工作目录下 cp dump2.rdb dump.rdb
- ◆ 启动 Redis, 备份数据会直接加载

### 12.2.9. 优势

- 适合大规模的数据恢复
- 对数据完整性和一致性要求不高更适合使用
- 节省磁盘空间
- 恢复速度快



### 12.2.10. 劣势

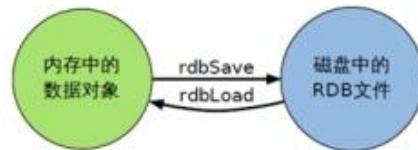
- Fork 的时候，内存中的数据被克隆了一份，大致 2 倍的膨胀性需要考虑
- 虽然 Redis 在 fork 时使用了写时拷贝技术，但是如果数据庞大时还是比较消耗性能。
- 在备份周期在一定间隔时间做一次备份，所以如果 Redis 意外 down 掉的话，就会丢失最后一次快照后的所有修改。

### 12.2.11. 如何停止

动态停止 RDB: redis-cli config set save ""#save 后给空值，表示禁用保存策略

### 12.2.12. 小总结

## RDB



- |  |  |
|--|--|
| <ul style="list-style-type: none"><li>• RDB是一个非常紧凑的文件</li><li>• RDB在保存RDB文件时父进程唯一需要做的就是fork出一个子进程,接下来的工作全部由子进程来做,父进程不需要再做其他IO操作,所以RDB持久化方式可以最大化redis的性能.</li><li>• 与AOF相比,在恢复大的数据集的时候, RDB方式会更快一些.</li></ul> | <ul style="list-style-type: none"><li>• 数据丢失风险大</li><li>• RDB 需要经常fork子进程来保存数据集到硬盘上,当数据集比较大的时候,fork的过程是非常耗时的,可能会导致Redis在一些毫秒级不能相应客户端请求</li></ul> |
|--|--|

## 13. Redis 持久化之 AOF

### 13.1. AOF (Append Only File)

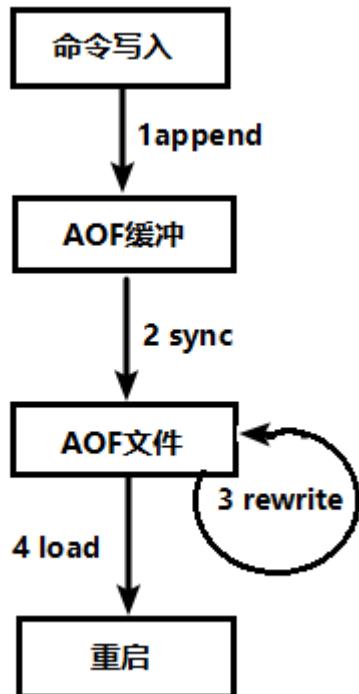
#### 13.1.1. 是什么

以日志的形式来记录每个写操作（增量保存），将 Redis 执行过的所有写指令记录下来（**读操作不记录**），**只许追加文件但不可以改写文件**，redis 启动之初会读取该文件重新构建数据，换言之，redis 重启的话就根据日志文件的内容将写指令从前到后执行一次以完成数据的恢复工作

#### 13.1.2. 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 sync 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;



### 13.1.3. 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。

### 13.1.4. AOF 和 RDB 同时开启, redis 听谁的?

AOF 和 RDB 同时开启, 系统默认取 AOF 的数据 (数据不会存在丢失)

### 13.1.5. AOF 启动/修复/恢复

- AOF 的备份机制和性能虽然和 RDB 不同, 但是备份和恢复的操作同 RDB 一样, 都是拷贝备份文件, 需要恢复时再拷贝到 Redis 工作目录下, 启动系统即加载。
- 正常恢复
  - 修改默认的 `appendonly no`, 改为 `yes`
  - 将有数据的 `aof` 文件复制一份保存到对应目录(查看目录: `config get dir`)

- 恢复：重启 redis 然后重新加载
- 异常恢复
  - 修改默认的 appendonly no，改为 yes
  - 如遇到 **AOF 文件损坏**，通过 /usr/local/bin/redis-check-aof--fix **appendonly.aof** 进行恢复
  - 备份被写坏的 AOF 文件
  - 恢复：重启 redis，然后重新加载

### 13.1.6. AOF 同步频率设置

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

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。

appendfsync no

redis 不主动进行同步，把**同步时机交给操作系统**。

### 13.1.7. Rewrite 压缩

1 是什么：

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

2 重写原理，如何实现重写

AOF 文件持续增长而过大时，会 fork 出一条新进程来将文件重写(也是先写临时文件最后再 rename)，redis4.0 版本后的重写，是指上就是把 rdb 的快照，以二级制的形式附在新的 aof 头部，作为已有的历史数据，替换掉原来的流水账操作。

`no-appendfsync-on-rewrite:`

如果 no-appendfsync-on-rewrite=yes ,不写入 aof 文件只写入缓存，用户请求不会阻塞，但是在这段时间如果宕机会丢失这段时间的缓存数据。 (降低数据安全性，提高性能)

如果 no-appendfsync-on-rewrite=no, 还是会把数据往磁盘里刷，但是遇到重写操作，可能会发生阻塞。 (数据安全，但是性能降低)

### 触发机制，何时重写

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

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。

auto-aof-rewrite-percentage：设置重写的基准值，文件达到 100% 时开始重写（文件是原来重写后文件的 2 倍时触发）

auto-aof-rewrite-min-size：设置重写的基准值，最小文件 64MB。达到这个值开始重写。

例如：文件达到 70MB 开始重写，降到 50MB，下次什么时候开始重写？100MB

系统载入时或者上次重写完毕时，Redis 会记录此时 AOF 大小，设为 base\_size，

如果 Redis 的 AOF 当前大小  $\geq \text{base\_size} + \text{base\_size} * 100\%$  (默认) 且当前大小  $\geq 64mb$  (默认) 的情况下，Redis 会对 AOF 进行重写。

### 3、重写流程

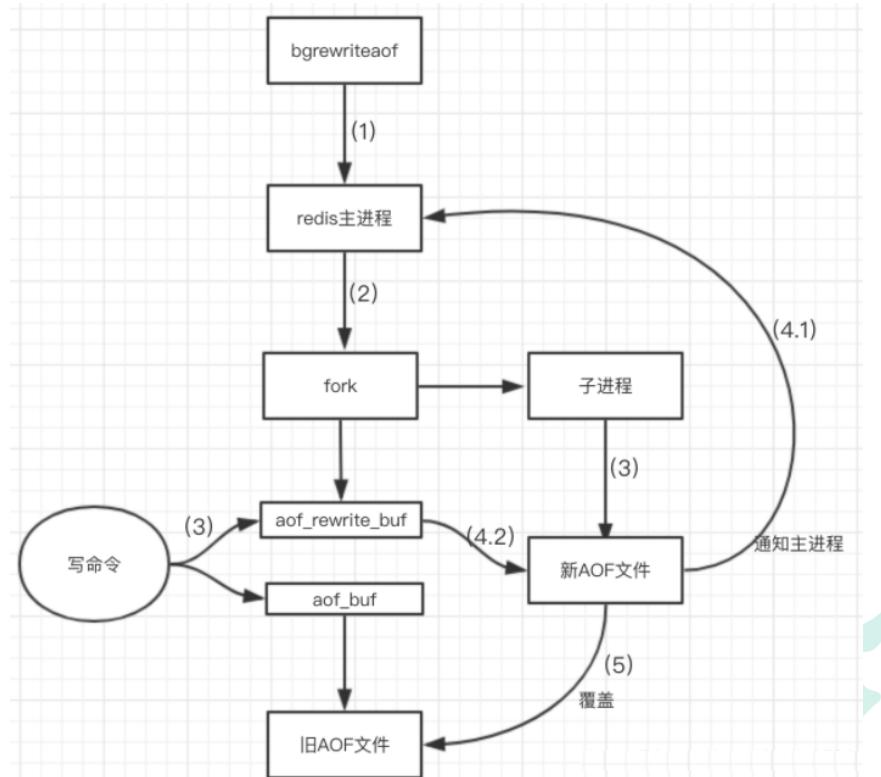
(1) bgsrewriteaof 触发重写，判断是否当前有 bgsave 或 bgsrewriteaof 在运行，如果有，则等待该命令结束后再继续执行。

(2) 主进程 fork 出子进程执行重写操作，保证主进程不会阻塞。

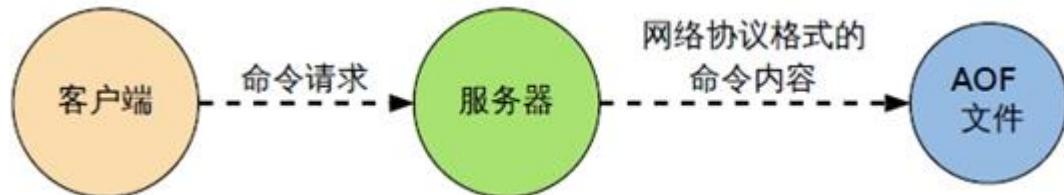
(3) 子进程遍历 redis 内存中数据到临时文件，客户端的写请求同时写入 aof\_buf 缓冲区和 aof\_rewrite\_buf 重写缓冲区保证原 AOF 文件完整以及新 AOF 文件生成期间的新数据修改动作不会丢失。

(4) 1).子进程写完新的 AOF 文件后，向主进程发信号，父进程更新统计信息。2).主进程把 aof\_rewrite\_buf 中的数据写入到新的 AOF 文件。

(5) 使用新的 AOF 文件覆盖旧的 AOF 文件，完成 AOF 重写。



### 13.1.8. 优势



- 备份机制更稳健，丢失数据概率更低。
- 可读的日志文本，通过操作 AOF 稳健，可以处理误操作。

### 13.1.9. 劣势

- 比起 RDB 占用更多的磁盘空间。
- 恢复备份速度要慢。
- 每次读写都同步的话，有一定的性能压力。
- 存在个别 Bug，造成恢复不能。

### 13.1.10. 小总结

## AOF



- AOF文件时一个只进行追加的日志文件
- Redis 可以在 AOF 文件体积变得过大时，自动地在后台对 AOF 进行重写
- AOF 文件有序地保存了对数据库执行的所有写入操作，这些写入操作以 Redis 协议的格式保存，因此 AOF 文件的内容非常容易被人读懂，对文件进行分析也很轻松
- 对于相同的数据集来说，AOF 文件的体积通常要大于 RDB 文件的体积
- 根据所使用的 fsync 策略，AOF 的速度可能会慢于 RDB

## 13.2. 总结(Which one)

### 13.2.1. 用哪个好

官方推荐两个都启用。

如果对数据不敏感，可以选单独用 RDB。

不建议单独用 AOF，因为可能会出现 Bug。

如果只是做纯内存缓存，可以都不用。

### 13.2.2. 官网建议

Ok, so what should I use?

The general indication is that you should use both persistence methods if you want a degree of data safety comparable to what PostgreSQL can provide you.

If you care a lot about your data, but still can live with a few minutes of data loss in case of disasters, you can simply use RDB alone.

There are many users using AOF alone, but we discourage it since to have an RDB snapshot from time to time is a great idea for doing database backups, for faster restarts, and in the event of bugs in the AOF engine.

Note: for all these reasons we'll likely end up unifying AOF and RDB into a single persistence model in the future (long term plan).

The following sections will illustrate a few more details about the two persistence models.

#### Snapshotting

By default Redis saves snapshots of the dataset on disk, in a binary file called `dump.rdb`. You can configure Redis to have it save the dataset every N seconds if there are at least M changes in the dataset, or you can manually call the `SAVE` or `BGSAVE` commands.

For example, this configuration will make Redis automatically dump the dataset to disk every 60 seconds if at least 1000 keys changed:

- RDB 持久化方式能够在指定的时间间隔能对你的数据进行快照存储
- AOF 持久化方式记录每次对服务器写的操作,当服务器重启的时候会重新执行这些命令来恢复原始的数据,AOF 命令以 redis 协议追加保存每次写的操作到文件末尾.
- Redis 还能对 AOF 文件进行后台重写,使得 AOF 文件的体积不至于过大
- 只做缓存：如果你只希望你的数据在服务器运行的时候存在,你也可以不使用任何持久化方式.
- 同时开启两种持久化方式
- 在这种情况下,当 redis 重启的时候会优先载入 AOF 文件来恢复原始的数据,因为在通常情况下 AOF 文件保存的数据集要比 RDB 文件保存的数据集要完整.
- RDB 的数据不实时, 同时使用两者时服务器重启也只会找 AOF 文件。那要不要只使用 AOF 呢？
- 建议不要, 因为 RDB 更适合用于备份数据库(AOF 在不断变化不好备份), 快速重启, 而且不会有 AOF 可能潜在的 bug, 留着作为一个万一的手段。
- 性能建议

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

如果使用 AOF，好处是在最恶劣情况下也只会丢失不超过两秒数据，启动脚本较简单只 load 自己的 AOF 文件就可以了。

代价，一是带来了持续的 IO，二是 AOF rewrite 的最后将 rewrite 过程中产生的新数据写到新文件造成的阻塞几乎是不可避免的。

只要硬盘许可，应该尽量减少 AOF rewrite 的频率，AOF 重写的基础大小默认值 **64M 太小了**，可以设到 **5G** 以上。

默认超过原大小 100% 大小时重写可以改到适当的数值。

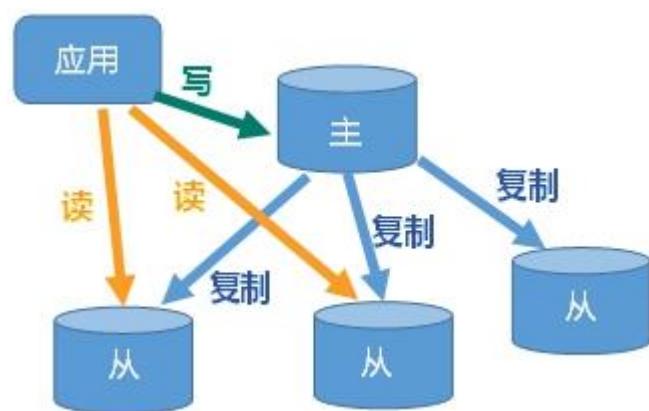
## 14. Redis\_主从复制

### 14.1. 是什么

主机数据更新后根据配置和策略，自动同步到备机的 **master/slaver 机制**，**Master 以写为主，Slave 以读为主**

### 14.2. 能干嘛

- 读写分离，性能扩展
- 容灾快速恢复



### 14.3. 怎么玩：主从复制

拷贝多个 redis.conf 文件 include(写绝对路径)

开启 daemonize yes

Pid 文件名字 pidfile

指定端口 port

Log 文件名字

dump.rdb 名字 dbfilename

Appendonly 关掉或者换名字

#### 14.3.1. 新建 redis6379.conf，填写以下内容

```
include /myredis/redis.conf
```

```
pidfile /var/run/redis_6379.pid
```

```
port 6379
```

```
dbfilename dump6379.rdb
```

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

#### 14.3.2. 新建 redis6380.conf，填写以下内容

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

#### 14.3.3. 新建 redis6381.conf，填写以下内容

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

slave-priority 10

设置从机的优先级，值越小，优先级越高，用于选举主机时使用。默认 100

#### 14.3.4. 启动三台 redis 服务器

```
[root@zy myredis]# redis-server redis6379.conf
[root@zy myredis]# redis-server redis6380.conf
[root@zy myredis]# redis-server redis6381.conf
```

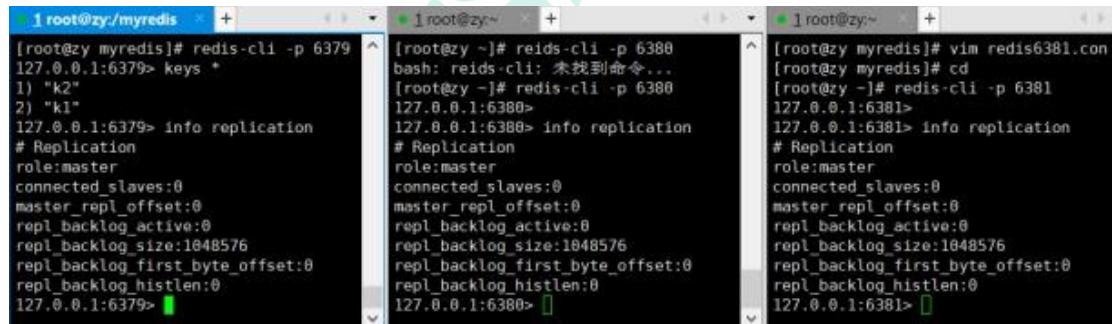
#### 14.3.5. 查看系统进程，看看三台服务器是否启动

```
[root@zy myredis]# ps -ef|grep redis
root      9640    7957  0 13:57 pts/1      00:00:00 vim redis.conf
root     10799      1  0 15:37 ?          00:00:00 redis-server *:6379
root     10803      1  0 15:37 ?          00:00:00 redis-server *:6380
root     10817      1  0 15:38 ?          00:00:00 redis-server *:6381
root     10821    7145  0 15:39 pts/3      00:00:00 grep --color=auto redis
```

#### 14.3.6. 查看三台主机运行情况

info replication

打印主从复制的相关信息



The image shows three separate terminal windows, each running a Redis command to check replication status. The first window (server 6379) shows it's a master with two slaves (6380 and 6381). The second window (server 6380) shows it's a slave connected to the master at 6379. The third window (server 6381) also shows it's a slave connected to the master at 6379.

```
[root@zy myredis]# redis-cli -p 6379
127.0.0.1:6379> keys *
1) "K2"
2) "K1"
127.0.0.1:6379> info replication
# Replication
role:master
connected_slaves:0
master_repl_offset:0
repl_backlog_active:0
repl_backlog_size:1048576
repl_backlog_first_byte_offset:0
repl_backlog_histlen:0
127.0.0.1:6379>

[root@zy ~]# redis-cli -p 6380
bash: redis-cli: 未找到命令...
[root@zy ~]# redis-cli -p 6380
127.0.0.1:6380> info replication
# Replication
role:master
connected_slaves:0
master_repl_offset:0
repl_backlog_active:0
repl_backlog_size:1048576
repl_backlog_first_byte_offset:0
repl_backlog_histlen:0
127.0.0.1:6380>

[root@zy myredis]# vim redis6381.conf
[root@zy myredis]# cd
[root@zy ~]# redis-cli -p 6381
127.0.0.1:6381>
127.0.0.1:6381> info replication
# Replication
role:master
connected_slaves:0
master_repl_offset:0
repl_backlog_active:0
repl_backlog_size:1048576
repl_backlog_first_byte_offset:0
repl_backlog_histlen:0
127.0.0.1:6381>
```

#### 14.3.7. 配从(库)不配主(库)

slaveof <ip><port>

成为某个实例的从服务器

1、在 6380 和 6381 上执行: slaveof 127.0.0.1 6379

```

[1] root@zy:~# redis-server /myredis/redis6379.conf
[1] root@zy:~# redis-cli -p 6379
127.0.0.1:6379> info replication
# Replication
role:master
connected_slaves:2
slave0:ip=127.0.0.1,port=6380,state=online,offset=57,lag=1
slave1:ip=127.0.0.1,port=6381,state=online,offset=57,lag=1
master_repl_offset:57
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:2
repl_backlog_histlen:56
127.0.0.1:6379> 

[2] 127.0.0.1:6380> slaveof 127.0.0.1 6379
OK
127.0.0.1:6380> info replication
# Replication
role:slave
master_host:127.0.0.1
master_port:6379
master_link_status:up
master_last_io_seconds_ago:9
master_sync_in_progress:0
slave_repl_offset:57
slave_priority:100
slave_read_only:1
connected_slaves:0
master_repl_offset:43
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:2
repl_backlog_histlen:42
127.0.0.1:6380> 

[3] 127.0.0.1:6381> slaveof 127.0.0.1 6379
OK
127.0.0.1:6381> info replication
# Replication
role:slave
master_host:127.0.0.1
master_port:6379
master_link_status:up
master_last_io_seconds_ago:3
master_sync_in_progress:0
slave_repl_offset:71
slave_priority:100
slave_read_only:1
connected_slaves:0
master_repl_offset:9
repl_backlog_active:0
repl_backlog_size:1048576
repl_backlog_first_byte_offset:0
repl_backlog_histlen:0
127.0.0.1:6381>

```

2、在主机上写，在从机上可以读取数据

在从机上写数据报错

```

127.0.0.1:6381> set k4 v4
(error) READONLY You can't write against a read only slave.

```

3、主机挂掉，重启就行，一切如初

4、从机重启需重设：slaveof 127.0.0.1 6379

可以将配置增加到文件中。永久生效。

```

[root@zy ~]# redis-server /myredis/redis6380.conf
[root@zy ~]# redis-cli -p 6380
127.0.0.1:6380> info replication
# Replication
role:master
connected_slaves:0
master_repl_offset:0
repl_backlog_active:0
repl_backlog_size:1048576
repl_backlog_first_byte_offset:0
repl_backlog_histlen:0

```

## 14.4. 常用 3 招

### 14.4.1. 一主二仆

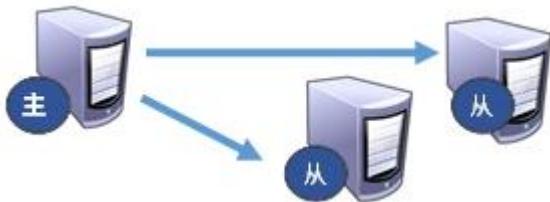
切入点问题？ slave1、slave2 是从头开始复制还是从切入点开始复制？比如从 k4 进来，那之前的 k1,k2,k3 是否也可以复制？

从机是否可以写？set 可否？

主机 shutdown 后情况如何？从机是上位还是原地待命？

主机又回来了后，主机新增记录，从机还能否顺利复制？

其中一台从机 down 后情况如何？依照原有它能跟上大部队吗？



#### 14.4.2. 薪火相传

上一个 Slave 可以是下一个 slave 的 Master，Slave 同样可以接收其他 slaves 的连接和同步请求，那么该 slave 作为链中下一个的 master，可以有效减轻 master 的写压力，去中心化降低风险。

用 `slaveof <ip><port>`

中途变更转向：会清除之前的数据，重新建立拷贝最新的

风险是一旦某个 slave 宕机，后面的 slave 都没法备份

主机挂了，从机还是从机，无法写数据了



```

127.0.0.1:6379> info replication
# Replication
role:master
connected_slaves:1
master_id:127.0.0.1:6380
state:online
offset:1915
lag:1
master_repl_offset:1915
replication_backlog_size:1048576
replication_backlog_active:0
replication_backlog_first_byte_offset:2
replication_backlog_histlen:1914
127.0.0.1:6379>

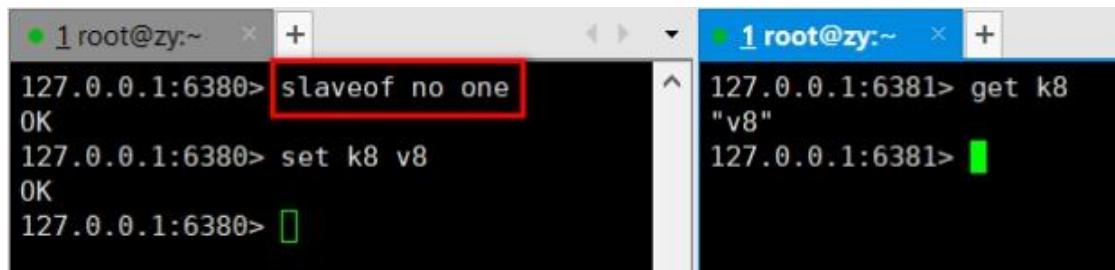
127.0.0.1:6380> info replication
# Replication
role:slave
master_id:127.0.0.1:6379
master_port:6379
master_link_status:up
master_link_id:second_slave_4
master_sync_in_progress:0
slave_repl_offset:1845
slave_priority:100
slave_rank:2
connected_slaves:1
slave0_id:127.0.0.1:6381
state:online
offset:43
lag:1
master_repl_offset:43
replication_backlog_size:1
replication_backlog_active:0
replication_backlog_first_byte_offset:2
replication_backlog_histlen:42
127.0.0.1:6380>

127.0.0.1:6381> info replication
# Replication
role:slave
master_id:127.0.0.1:6380
master_port:6380
master_link_status:up
master_link_id:third_slave_5
master_sync_in_progress:0
slave_priority:10
slave_read_only:1
connected_slaves:0
master_repl_offset:0
replication_backlog_size:0
replication_backlog_active:0
replication_backlog_first_byte_offset:0
replication_backlog_histlen:0
127.0.0.1:6381>
  
```

#### 14.4.3. 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。

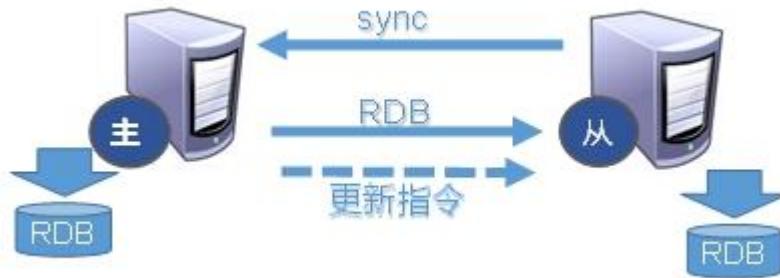
用 slaveof no one 将从机变为主机。



```
127.0.0.1:6380> slaveof no one
OK
127.0.0.1:6380> set k8 v8
OK
127.0.0.1:6380>
127.0.0.1:6381> get k8
"v8"
127.0.0.1:6381>
```

## 14.5. 复制原理

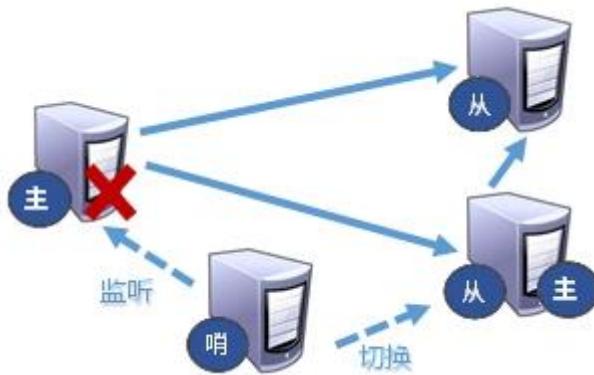
- Slave 启动成功连接到 master 后会发送一个 sync 命令
- Master 接到命令启动后台的存盘进程，同时收集所有接收到的用于修改数据集命令，在后台进程执行完毕之后，master 将传送整个数据文件到 slave，以完成一次完全同步
- 全量复制：而 slave 服务在接收到数据库文件数据后，将其存盘并加载到内存中。
- 增量复制：Master 继续将新的所有收集到的修改命令依次传给 slave，完成同步
- 但是只要是重新连接 master，一次完全同步（全量复制）将被自动执行



## 14.6. 哨兵模式(sentinel)

### 14.6.1. 是什么

**反客为主的自动版**，能够后台监控主机是否故障，如果故障了根据投票数自动将从库转换为主库



## 14.6.2. 怎么玩(使用步骤)

### 14.6.2.1. 调整为一主二仆模式，6379 带着 6380、6381

```

[1] root@zy:~/myredis# redis-server redis6379.conf
[1] root@zy:~/myredis# redis-cli -p 6379
127.0.0.1:6379> info replication
# Replication
role:master
connected_slaves:2
slave0:ip=127.0.0.1,port=6380,state=online,offset=57,lag=1
slave1:ip=127.0.0.1,port=6381,state=online,offset=57,lag=1
master_repl_offset:57
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:2
repl_backlog_histlen:56
127.0.0.1:6379> [1]
[2] 127.0.0.1:6380> slaveof 127.0.0.1 6379
OK
127.0.0.1:6380> info replication
# Replication
role:slave
master_host:127.0.0.1
master_port:6379
master_link_status:up
master_last_io_seconds_ago:9
master_sync_in_progress:0
slave_repl_offset:57
slave_priority:100
slave_read_only:1
connected_slaves:0
master_repl_offset:43
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:2
repl_backlog_histlen:42
127.0.0.1:6380> [2]
[3] 127.0.0.1:6381> slaveof 127.0.0.1 6379
OK
127.0.0.1:6381> info replication
# Replication
role:slave
master_host:127.0.0.1
master_port:6379
master_link_status:up
master_last_io_seconds_ago:3
master_sync_in_progress:0
slave_repl_offset:71
slave_priority:10
slave_read_only:1
connected_slaves:0
master_repl_offset:0
repl_backlog_active:0
repl_backlog_size:1048576
repl_backlog_first_byte_offset:0
repl_backlog_histlen:0
127.0.0.1:6381> [3]

```

### 14.6.2.2. 自定义的/myredis 目录下新建 sentinel.conf 文件，名字 绝不能错

### 14.6.2.3. 配置哨兵,填写内容

sentinel monitor mymaster 127.0.0.1 6379 1

其中 mymaster 为监控对象起的服务器名称， 1 为至少有多少个哨兵同意迁移的数量。

### 14.6.2.4. 启动哨兵

/usr/local/bin

redis 做压测可以用自带的 **redis-benchmark** 工具

执行 **redis-sentinel /myredis/sentinel.conf**

```
[root@zy myredis]# ll /usr/local/bin
总用量 15064
-rw-r--r--. 1 root root    266 12月 20 19:19 dump.rdb
-rwxr-xr-x. 1 root root 2432464 12月 16 12:28 redis-benchmark
-rwxr-xr-x. 1 root root   25168 12月 16 12:28 redis-check-aof
-rwxr-xr-x. 1 root root 5182944 12月 16 12:28 redis-check-rdb
-rwxr-xr-x. 1 root root 2585176 12月 16 12:28 redis-cli
lrwxrwxrwx. 1 root root      12 12月 16 12:28 redis-sentinel -> redis-server
-rwxr-xr-x. 1 root root 5182944 12月 16 12:28 redis-server
[root@zy myredis]# redis-sentinel /myredis/sentinel.conf
11738:X 21 Dec 16:55:06.504 * Increased maximum number of open files to 10032 (it
was originally set to 1024).

                               Redis 3.2.5 (00000000/0) 64 bit
                               Running in sentinel mode
                               Port: 26379
                               PID: 11738

                               http://redis.io

11738:X 21 Dec 16:55:06.505 # WARNING: The TCP backlog setting of 511 cannot be en
forced because /proc/sys/net/core/somaxconn is set to the lower value of 128.
11738:X 21 Dec 16:55:06.507 # Sentinel ID is 6c485654c0ef00ba073f2581b7aa304de1862
774
11738:X 21 Dec 16:55:06.507 # +monitor master mymaster 127.0.0.1 6379 quorum 1
```

#### 14.6.2.5. 当主机挂掉，从机选举中产生新的主机

(大概 10 秒左右可以看到哨兵窗口日志，切换了新的主机)

哪个从机会被选举为主机呢？根据优先级别：slave-priority

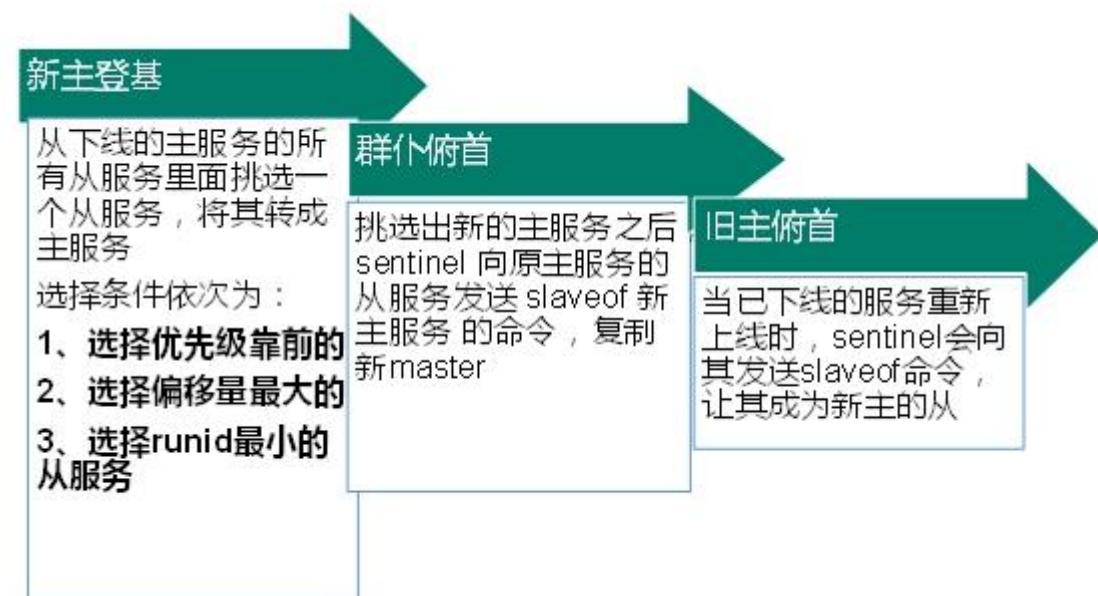
原主机重启后会变为从机。

```
11897:X 21 Dec 17:07:02.380 # +sdown master mymaster 127.0.0.1 6379
11897:X 21 Dec 17:07:02.380 # +odown master mymaster 127.0.0.1 6379 #quorum 1/1
11897:X 21 Dec 17:07:02.380 # +new-epoch 2
11897:X 21 Dec 17:07:02.380 # +try-failover master mymaster 127.0.0.1 6379
11897:X 21 Dec 17:07:02.384 # +vote-for-leader 6c485654c0ef00ba073f2581b7aa304de18
62774 2
11897:X 21 Dec 17:07:02.384 # +elected-leader master mymaster 127.0.0.1 6379
11897:X 21 Dec 17:07:02.384 # +failover-state-select-slave master mymaster 127.0.0.
.1 6379
11897:X 21 Dec 17:07:02.446 # +selected-slave slave 127.0.0.1:6381 127.0.0.1 6381
@ mymaster 127.0.0.1 6379
11897:X 21 Dec 17:07:02.446 * +failover-state-send-slaveof-noone slave 127.0.0.1:6
381 127.0.0.1 6381 @ mymaster 127.0.0.1 6379
11897:X 21 Dec 17:07:02.547 * +failover-state-wait-promotion slave 127.0.0.1:6381
127.0.0.1 6381 @ mymaster 127.0.0.1 6379
11897:X 21 Dec 17:07:02.609 # +promoted-slave slave 127.0.0.1:6381 127.0.0.1 6381
@ mymaster 127.0.0.1 6379
11897:X 21 Dec 17:07:02.609 # +failover-state-reconf-slaves master mymaster 127.0.
.1 6379
11897:X 21 Dec 17:07:02.707 * +slave-reconf-sent slave 127.0.0.1:6380 127.0.0.1 63
80 @ mymaster 127.0.0.1 6379
11897:X 21 Dec 17:07:03.628 * +slave-reconf-inprog slave 127.0.0.1:6380 127.0.0.1
6380 @ mymaster 127.0.0.1 6379
11897:X 21 Dec 17:07:03.628 * +slave-reconf-done slave 127.0.0.1:6380 127.0.0.1 63
80 @ mymaster 127.0.0.1 6379
11897:X 21 Dec 17:07:03.728 # +failover-end master mymaster 127.0.0.1 6379
11897:X 21 Dec 17:07:03.728 # +switch-master mymaster 127.0.0.1 6379 127.0.0.1 638
1
11897:X 21 Dec 17:07:03.728 * +slave slave 127.0.0.1:6380 127.0.0.1 6380 @ mymaste
r 127.0.0.1 6381
11897:X 21 Dec 17:07:03.728 * +slave slave 127.0.0.1:6379 127.0.0.1 6379 @ mymaste
r 127.0.0.1 6381
11897:X 21 Dec 17:07:33.772 # +sdown slave 127.0.0.1:6379 127
.0.0.1 6379 @ mymaster 127.0.0.1 6381
```

#### 14.6.2.6. 复制延时

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，当系统很繁忙的时候，延迟问题会更加严重，Slave 机器数量的增加也会使这个问题更加严重。

### 14.6.3. 故障恢复



优先级在 redis.conf 中默认：slave-priority 100，值越小优先级越高

偏移量是指获得原主机数据最全的

每个 redis 实例启动后都会随机生成一个 40 位的 runid

### 14.6.4. 主从复制

```
private static JedisSentinelPool jedisSentinelPool=null;

public static Jedis getJedisFromSentinel() {
    if(jedisSentinelPool==null) {
        Set<String> sentinelSet=new HashSet<>();
        sentinelSet.add("192.168.11.103:26379");

        JedisPoolConfig jedisPoolConfig =new JedisPoolConfig();
        jedisPoolConfig.setMaxTotal(10); //最大可用连接数
        jedisPoolConfig.setMaxIdle(5); //最大闲置连接数
        jedisPoolConfig.setMinIdle(5); //最小闲置连接数
        jedisPoolConfig.setBlockWhenExhausted(true); //连接耗尽是否等待
        jedisPoolConfig.setMaxWaitMillis(2000); //等待时间
        jedisPoolConfig.setTestOnBorrow(true); //取连接的时候进行一下测试 ping pong

        jedisSentinelPool=new
        JedisSentinelPool("mymaster", sentinelSet, jedisPoolConfig);
        return jedisSentinelPool.getResource();
    }
}
```

```
        } else {
    return jedisSentinelPool.getResource();
}
}
```

## 15. Redis 集群

### 15.1. 问题

容量不够，redis 如何进行扩容？

并发写操作，redis 如何分摊？

另外，主从模式，薪火相传模式，主机宕机，导致 ip 地址发生变化，应用程序中配置需要修改对应的主机地址、端口等信息。

之前通过代理主机来解决，但是 redis3.0 中提供了解决方案。就是无中心化集群配置。

### 15.2. 什么是集群

Redis 集群实现了对 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 个节点中，每个节点存储总数据的 1/N。

Redis 集群通过分区（partition）来提供一定程度的可用性（availability）：即使集群中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请求。

### 15.3. 删除持久化数据

将 .rdb,.aof 文件都删除掉。

### 15.4. 制作 6 个实例，6379,6380,6381,6389,6390,6391

#### 15.4.1. 配置基本信息

开启 daemonize yes

Pid 文件名字

指定端口

Log 文件名字

### Dump.rdb 名字

Appendonly 关掉或者换名字

### 15.4.2. redis cluster 配置修改

cluster-enabled yes 打开集群模式

cluster-config-file nodes-6379.conf 设定节点配置文件名

cluster-node-timeout 15000 设定节点失联时间，超过该时间（毫秒），集群自动进行主从切换。

```
include /home/bigdata/redis.conf
port 6379
pidfile "/var/run/redis_6379.pid"
dbfilename "dump6379.rdb"
dir "/home/bigdata/redis_cluster"
logfile "/home/bigdata/redis_cluster/redis_err_6379.log"
cluster-enabled yes
cluster-config-file nodes-6379.conf
cluster-node-timeout 15000
```

### 15.4.3. 修改好 redis6379.conf 文件，拷贝多个 redis.conf 文件

```
redis6379.conf
redis6380.conf
redis6381.conf
redis6389.conf
redis6390.conf
redis6391.conf
```

#### 15.4.4. 使用查找替换修改另外 5 个文件

例如: :%s/6379/6380

#### 15.4.5. 启动 6 个 redis 服务

```
[root@zy myredis]# ps -ef|grep redis
root      14256  13631  0 20:59 pts/1    00:00:00 grep --color=auto redis
[root@zy myredis]# ll
总用量 76
-rw-r--r--. 1 root root   181 12月 21 20:50 redis6379.conf
-rw-r--r--. 1 root root   181 12月 21 20:51 redis6380.conf
-rw-r--r--. 1 root root   181 12月 21 20:51 redis6381.conf
-rw-r--r--. 1 root root   181 12月 21 20:51 redis6389.conf
-rw-r--r--. 1 root root   181 12月 21 20:52 redis6390.conf
-rw-r--r--. 1 root root   181 12月 21 20:52 redis6391.conf
-rw-r--r--. 1 root root  46705 12月 21 13:57 redis.conf
-rw-r--r--. 1 root root   335 12月 21 17:07 sentinel.conf
[root@zy myredis]# redis-server redis6379.conf
[root@zy myredis]# redis-server redis6380.conf
[root@zy myredis]# redis-server redis6381.conf
[root@zy myredis]# redis-server redis6389.conf
[root@zy myredis]# redis-server redis6390.conf
[root@zy myredis]# redis-server redis6391.conf
[root@zy myredis]# ps -ef|grep redis
root      14261      1  0 20:59 ?    00:00:00 redis-server *:6379 [cluster]
root      14265      1  0 20:59 ?    00:00:00 redis-server *:6380 [cluster]
root      14269      1  0 20:59 ?    00:00:00 redis-server *:6381 [cluster]
root      14273      1  0 20:59 ?    00:00:00 redis-server *:6389 [cluster]
root      14277      1  0 20:59 ?    00:00:00 redis-server *:6390 [cluster]
root      14281      1  0 20:59 ?    00:00:00 redis-server *:6391 [cluster]
root      14285  13631  0 20:59 pts/1    00:00:00 grep --color=auto redis
```

### 15.5. 将六个节点合成一个集群

组合之前, 请确保所有 redis 实例启动后, nodes-xxxx.conf 文件都生成正常。

```
[root@zy myredis]# ll
总用量 100
-rw-r--r--. 1 root root      0 12月 21 20:59 appendonly.aof
-rw-r--r--. 1 root root    112 12月 21 20:59 nodes-6379.conf
-rw-r--r--. 1 root root    112 12月 21 20:59 nodes-6380.conf
-rw-r--r--. 1 root root    112 12月 21 20:59 nodes-6381.conf
-rw-r--r--. 1 root root    112 12月 21 20:59 nodes-6389.conf
-rw-r--r--. 1 root root    112 12月 21 20:59 nodes-6390.conf
-rw-r--r--. 1 root root    112 12月 21 20:59 nodes-6391.conf
-rw-r--r--. 1 root root    181 12月 21 20:50 redis6379.conf
-rw-r--r--. 1 root root    181 12月 21 20:51 redis6380.conf
-rw-r--r--. 1 root root    181 12月 21 20:51 redis6381.conf
-rw-r--r--. 1 root root    181 12月 21 20:51 redis6389.conf
-rw-r--r--. 1 root root    181 12月 21 20:52 redis6390.conf
-rw-r--r--. 1 root root    181 12月 21 20:52 redis6391.conf
-rw-r--r--. 1 root root 46705 12月 21 13:57 redis.conf
-rw-r--r--. 1 root root    335 12月 21 17:07 sentinel.conf
```

- 合体：

```
cd /opt/redis-6.2.1/src
```

```
redis-cli --cluster create --cluster-replicas 1 192.168.11.101:6379
192.168.11.101:6380 192.168.11.101:6381 192.168.11.101:6389
192.168.11.101:6390 192.168.11.101:6391
```

此处不要用 127.0.0.1， 请用真实 IP 地址

--replicas 1 采用最简单的方式配置集群，一台主机，一台从机，正好三组。

```
[root@zy src]# ./redis-trib.rb create --replicas 1 192.168.137.3:6379 192.168.137.3:6380 192.168.137.3:6381 192.168.137.3:6389 192.168.137.3:6390 192.168.137.3:6391
>>> Creating cluster
>>> Performing hash slots allocation on 6 nodes...
Using 3 masters:
192.168.137.3:6379
192.168.137.3:6380
192.168.137.3:6381
Adding replica 192.168.137.3:6389 to 192.168.137.3:6379
Adding replica 192.168.137.3:6390 to 192.168.137.3:6389
Adding replica 192.168.137.3:6391 to 192.168.137.3:6381
M: b87514db03d13bd2e2a03c007073839f4d7fe3d 192.168.137.3:6379
  slots:0-5468 (5468 slots) master
M: 53d5a286c739d2c61bfa9a546d5704d25c23a8#4 192.168.137.3:6380
  slots:5469-10922 (5463 slots) master
M: 3aa5d4403b16e50fb17e9ca67f5ec1d61f484a 192.168.137.3:6381
  slots:10923-16383 (5461 slots) master
S: 964d024974aee8112945e01b79b564422ca818 192.168.137.3:6389
  replicates b87514db03d13bd2e2a03c007073839f4d7fe3d
S: 93bc38767888f53d97da2a8081068a091c14d4 192.168.137.3:6390
  replicates 53d5a286c739d2c61bfa9a546d5784d25c23a8#4
S: bc7099e2376714b9a99fc13f928faabbbee4cbf21 192.168.137.3:6391
  replicates 3aa5d4403b16e50fb17e9ca67f5ec3d61f484a
Can I set the above configuration? (type 'yes' to accept): yes
```

```
>>> Nodes configuration updated
>>> Assign a different config epoch to each node
>>> Sending CLUSTER MEET messages to join the cluster
Waiting for the cluster to join....
>>> Performing Cluster Check (using node 192.168.137.3:6379)
M: b87514db034d338d2e2a03c007073839f4d7fe3d 192.168.137.3:6379
  slots:0-5460 (5461 slots) master
    1 additional replica(s)
S: bc7090e237d714b9a09fc13f928fabbbbe4cbf21 192.168.137.3:6391
  slots: (0 slots) slave
    replicates 3aa5d4403b16be50fbe17e9ca67f5ec3d61f484a
S: 964d024974ac2be8112945eb1b79b564422ca818 192.168.137.3:6389
  slots: (0 slots) slave
    replicates b87514db034d338d2e2a03c007073839f4d7fe3d
S: 93bc38767868bf53d97da2aa8c81868a091c14d4 192.168.137.3:6390
  slots: (0 slots) slave
    replicates 53d5a286c739d2c61bfa9a546d5704d25c23a8e4
M: 53d5a286c739d2c61bfa9a546d5704d25c23a8e4 192.168.137.3:6380
  slots:5461-10922 (5462 slots) master
    1 additional replica(s)
M: 3aa5d4403b16be50fbe17e9ca67f5ec3d61f484a 192.168.137.3:6381
  slots:10923-16383 (5461 slots) master
    1 additional replica(s)
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.
```

- 普通方式登录

可能直接进入读主机，存储数据时，会出现 MOVED 重定向操作。所以，应该以集群方式登录。

```
[root@zy src]# redis-cli -p 6379
127.0.0.1:6379> keys *
(empty list or set)
127.0.0.1:6379> set k1 v1
(error) MOVED 12706 192.168.137.3:6381
```

## 15.6. -c 采用集群策略连接，设置数据会自动切换到相应的

### 写主机

```
[root@zy src]# redis-cli -c -p 6379
127.0.0.1:6379> keys *
(empty list or set)
127.0.0.1:6379> set k1 v1
-> Redirected to slot [12706] located at 192.168.137.3:6381
OK
192.168.137.3:6381>
```

## 15.7. 通过 cluster nodes 命令查看集群信息

```
192.168.137.3:6380> cluster nodes
+3d5a28fc739d2c61bfa9a546d784d25c23a8e4 192.168.137.3:6380 master - 0 1545398222136 2 connected 5461-10922
+aa5d4403b16ae0fbe17e9c467f5ec3d611484a 192.168.137.3:6380 myself,master - 0 0 3 connected 10923-16383
+b87514db034d38d2e2a03c087073839fd7fe3d 192.168.137.3:6379 master - 0 1545398221127 1 connected 0-5460
+bc7698e23/d714b9ad9fc131928fbabbe0e4cbf21 192.168.137.3:6391 slave +aa5d4403b16ae0fbe17e9c467f5ec3d611484a 0 1545398220119 6 connected
+964d924974ac2be8112945eb1b79b56422ca818 192.168.137.3:6389 slave +b87514db034d38d2e2a03c087073839fd7fe3d 0 1545398210105 4 connected
+930c387678680f53d97da2aa8c81868a091c14d4 192.168.137.3:6390 slave +5d5a28fc739d2c61bfa9a546d5784d25c23a8e4 0 1545398219112 5 connected
```

## 15.8. redis cluster 如何分配这六个节点?

一个集群至少要有**三个主节点**。

选项 --cluster-replicas 1 表示我们希望为集群中的每个主节点创建一个从节点。

分配原则尽量保证每个主数据库运行在不同的 IP 地址，每个从库和主库不在一个 IP 地址上。

## 15.9. 什么是 slots

[OK] All nodes agree about slots configuration.

>>> Check for open slots...

>>> Check slots coverage...

**[OK] All 16384 slots covered.**

一个 Redis 集群包含 16384 个插槽 (hash slot)， 数据库中的每个键都属于这 16384 个插槽的其中一个，

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插槽。 举个例子， 如果一个集群可以有主节点， 其中：

节点 A 负责处理 0 号至 5460 号插槽。

节点 B 负责处理 5461 号至 10922 号插槽。

节点 C 负责处理 10923 号至 16383 号插槽。

## 15.10. 在集群中录入值

在 redis-cli 每次录入、查询键值，redis 都会计算出该 key 应该送往的插槽，如果不是该客户端对应服务器的插槽，redis 会报错，并告知应前往的 redis 实例地址和端口。

redis-cli 客户端提供了 **-c** 参数实现自动重定向。

如 **redis-cli -c -p 6379** 登入后，再录入、查询键值对可以自动重定向。

不在一个 slot 下的键值，是不能使用 mget,mset 等多键操作。

```
192.168.137.3:6381> mset k1 v1 k2 v2 k3 v3  
(error) CROSS SLOT Keys in request don't hash to the same slot
```

可以通过{}来定义组的概念，从而使 key 中{}内相同内容的键值对放到一个 slot 中去。

```
192.168.137.3:6381> mset k1{cust} v1 k2{cust} v2 k3{cust} v3  
-> Redirected to slot [4847] located at 192.168.137.3:6379  
OK
```

## 15.11. 查询集群中的值

CLUSTER GETKEYSINSLOT <slot><count> 返回 count 个 slot 槽中的键。

```
192.168.137.3:6379> cluster keyslot cust  
(integer) 4847  
192.168.137.3:6379> cluster countkeysinslot 4847  
(integer) 3  
192.168.137.3:6379> cluster getkeysinslot 4847 10  
1) "k1{cust}"  
2) "k2{cust}"  
3) "k3{cust}"
```

## 15.12. 故障恢复

如果主节点下线？从节点能否自动升为主节点？注意：15 秒超时

```
127.0.0.1:6380> cluster nodes  
b87514db034d38bd2e2a03c007073839f4d7fe3d 192.168.137.3:6379 [master, fail] - 1545399235089 1545399231958 1 disconnected  
964d024974ac2be8112945eb1b79b564422ca818 192.168.137.3:6389 [master] - 0 1545399381305 7 connected 0-5460  
bc7090e237d714b9a09fc13f928fabbbcc4cbf21 192.168.137.3:6391 [slave] 3aa5d4403b16be50fbef17e9ca67f5ec3d61f484a 0 1545399382313 6 connected  
53d5a286c739d2c61bf9a94546d784d25c23a8e4 192.168.137.3:6388 [myself, master] - 0 0 2 connected 5461-10922  
93bc38767868bf53d97da2ea8c1868a091c14d4 192.168.137.3:6390 [slave] 53d5a286c739d2c61bf9a546d5704d25c23a8e4 0 1545399383320 5 connected  
3aa5d4403b16be50fbef17e9ca67f5ec3d61f484a 192.168.137.3:6381 [master] - 0 1545399379288 3 connected 10923-16383
```

主节点恢复后，主从关系会如何？主节点回来变成从机。

```
127.0.0.1:6380> cluster nodes  
b87514db034d38bd2e2a03c007073839f4d7fe3d 192.168.137.3:6379 [slave] 964d024974ac2be8112945eb1b79b564422ca818 0 1545399515482 7 connected  
964d024974ac2be8112945eb1b79b564422ca818 192.168.137.3:6389 [master] - 0 1545399513467 7 connected 0-5460  
bc7090e237d714b9a09fc13f928fabbbcc4cbf21 192.168.137.3:6391 [slave] 3aa5d4403b16be50fbef17e9ca67f5ec3d61f484a 0 1545399511450 6 connected  
53d5a286c739d2c61bf9a546d5784d25c23a8e4 192.168.137.3:6388 [myself, master] - 0 0 2 connected 5461-10922  
93bc38767868bf53d97da2ea8c1868a091c14d4 192.168.137.3:6390 [slave] 53d5a286c739d2c61bf9a546d5704d25c23a8e4 0 1545399516488 5 connected  
3aa5d4403b16be50fbef17e9ca67f5ec3d61f484a 192.168.137.3:6381 [master] - 0 1545399514474 3 connected 10923-16383
```

如果所有某一段插槽的主从节点都宕掉，redis 服务是否还能继续？

如果某一段插槽的主从都挂掉，而 cluster-require-full-coverage 为 yes，那么，整个集群都挂掉

如果某一段插槽的主从都挂掉，而 cluster-require-full-coverage 为 no，那么，该插槽数据全都不能使用，也无法存储。

redis.conf 中的参数 cluster-require-full-coverage

## 15.13. 集群的 Jedis 开发

即使连接的不是主机，集群会自动切换主机存储。主机写，从机读。

无中心化主从集群。无论从哪台主机写的数据，其他主机上都能读到数据。

```
public class JedisClusterTest {  
    public static void main(String[] args) {  
        Set<HostAndPort> set = new HashSet<HostAndPort>();  
        set.add(new HostAndPort("192.168.31.211", 6379));  
        JedisCluster jedisCluster = new JedisCluster(set);  
        jedisCluster.set("k1", "v1");  
        System.out.println(jedisCluster.get("k1"));  
    }  
}
```

## 15.14. Redis 集群提供了以下好处

实现扩容

分摊压力

无中心配置相对简单

## 15.15. Redis 集群的不足

多键操作是不被支持的

多键的 Redis 事务是不被支持的。lua 脚本不被支持

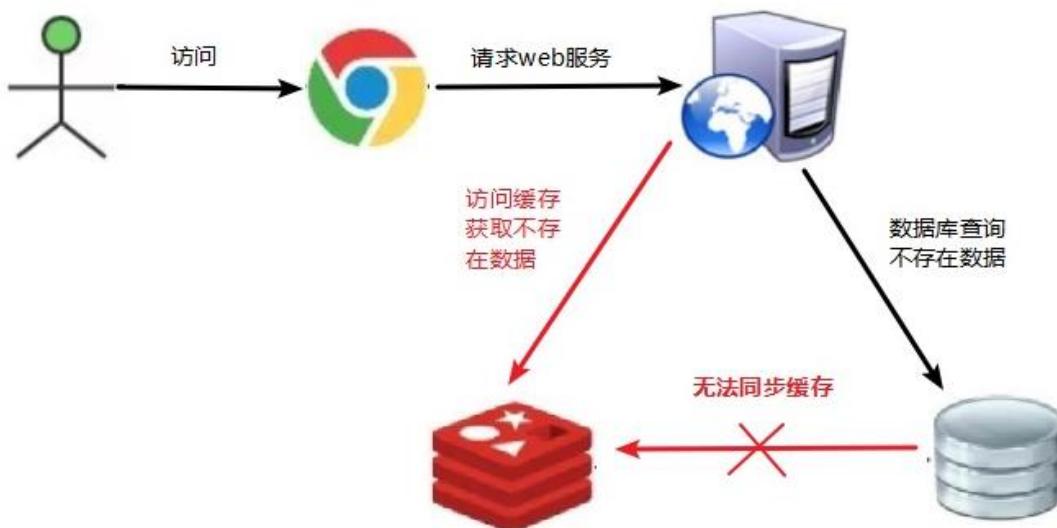
由于集群方案出现较晚，很多公司已经采用了其他的集群方案，而代理或者客户端分片的方案想要迁移至 redis cluster，需要整体迁移而不是逐步过渡，复杂度较大。

# 16. Redis 应用问题解决

## 16.1. 缓存穿透

### 16.1.1. 问题描述

key 对应的数据在数据源并不存在，每次针对此 key 的请求从缓存获取不到，请求都会压到数据源，从而可能压垮数据源。比如用一个不存在的用户 id 获取用户信息，不论缓存还是数据库都没有，若黑客利用此漏洞进行攻击可能压垮数据库。



### 16.1.2. 解决方案

一个一定不存在缓存及查询不到的数据，由于缓存是不命中时被动写的，并且出于容错考虑，如果从存储层查不到数据则不写入缓存，这将导致这个不存在的数据每次请求都要到存储层去查询，失去了缓存的意义。

解决方案：

- (1) **对空值缓存：**如果一个查询返回的数据为空（不管是数据是否存在），我们仍然把这个空结果（null）进行缓存，设置空结果的过期时间会很短，最长不超过五分钟
- (2) **设置可访问的名单（白名单）：**  
使用 bitmaps 类型定义一个可以访问的名单，名单 id 作为 bitmaps 的偏移量，每次访问和 bitmap 里面的 id 进行比较，如果访问 id 不在 bitmaps 里面，进行拦截，不允许访问。
- (3) **采用布隆过滤器：**（布隆过滤器（Bloom Filter）是 1970 年由布隆提出的。它实际上是一个很长的二进制向量（位图）和一系列随机映射函数（哈希函数）。

布隆过滤器可以用于检索一个元素是否在一个集合中。它的优点是空间效率和查询时间都远远超过一般的算法，缺点是有一定的误识别率和删除困难。)

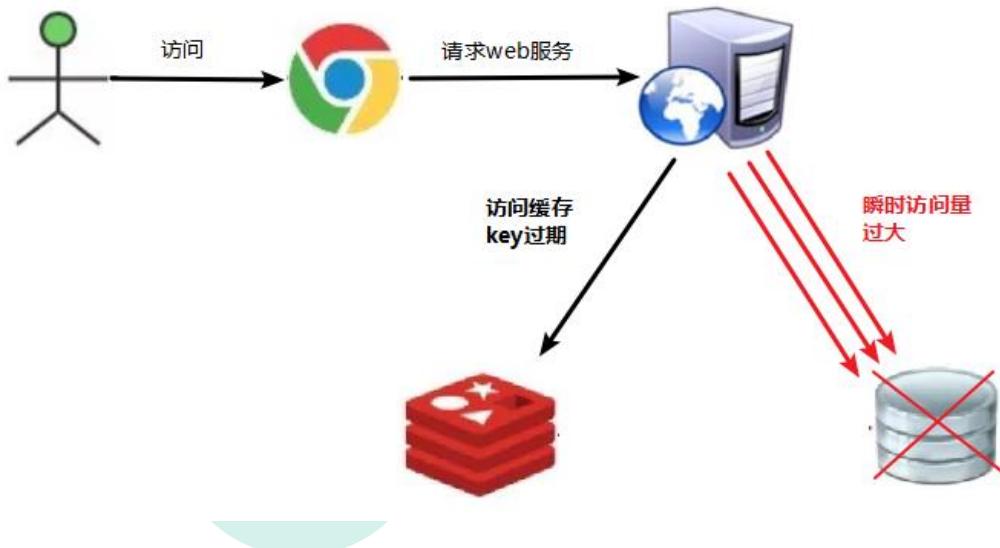
将所有可能存在的数据哈希到一个足够大的 bitmaps 中，一个一定不存在的数据会被这个 bitmaps 拦截掉，从而避免了对底层存储系统的查询压力。

- (4) 进行实时监控：**当发现 Redis 的命中率开始急速降低，需要排查访问对象和访问的数据，和运维人员配合，可以设置黑名单限制服务

## 16.2. 缓存击穿

### 16.2.1. 问题描述

key 对应的数据存在，但在 redis 中过期，此时若有大量并发请求过来，这些请求发现缓存过期一般都会从后端 DB 加载数据并回写到缓存，这个时候大并发的请求可能会瞬间把后端 DB 压垮。



### 16.2.2. 解决方案

key 可能在某些时间点被超高并发地访问，是一种非常“热点”的数据。这个时候，需要考虑一个问题：缓存被“击穿”的问题。

解决问题：

**(1) 预先设置热门数据：**在 redis 高峰访问之前，把一些热门数据提前存入到 redis 里面，加大这些热门数据 key 的时长

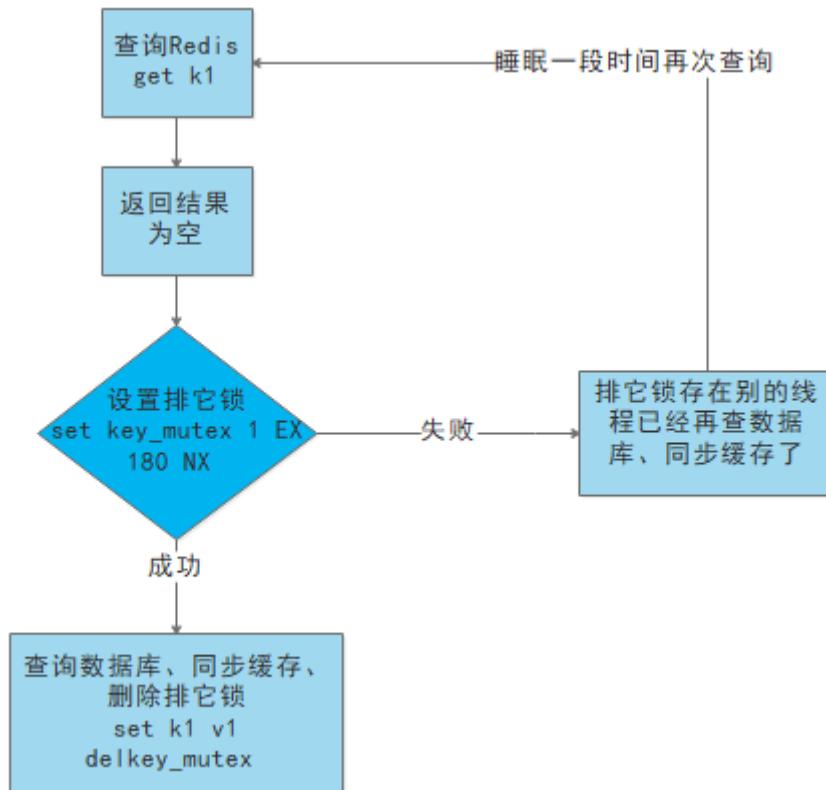
**(2) 实时调整：**现场监控哪些数据热门，实时调整 key 的过期时长

**(3) 使用锁：**

(1) 就是在缓存失效的时候（判断拿出来的值为空），不是立即去 load db。

(2) 先使用缓存工具的某些带成功操作返回值的操作（比如 Redis 的 SETNX）

- 去 set 一个 mutex key
- (3) 当操作返回成功时，再进行 load db 的操作，并回设缓存,最后删除 mutex key;
- (4) 当操作返回失败，证明有线程在 load db，当前线程睡眠一段时间再重试整个 get 缓存的方法。



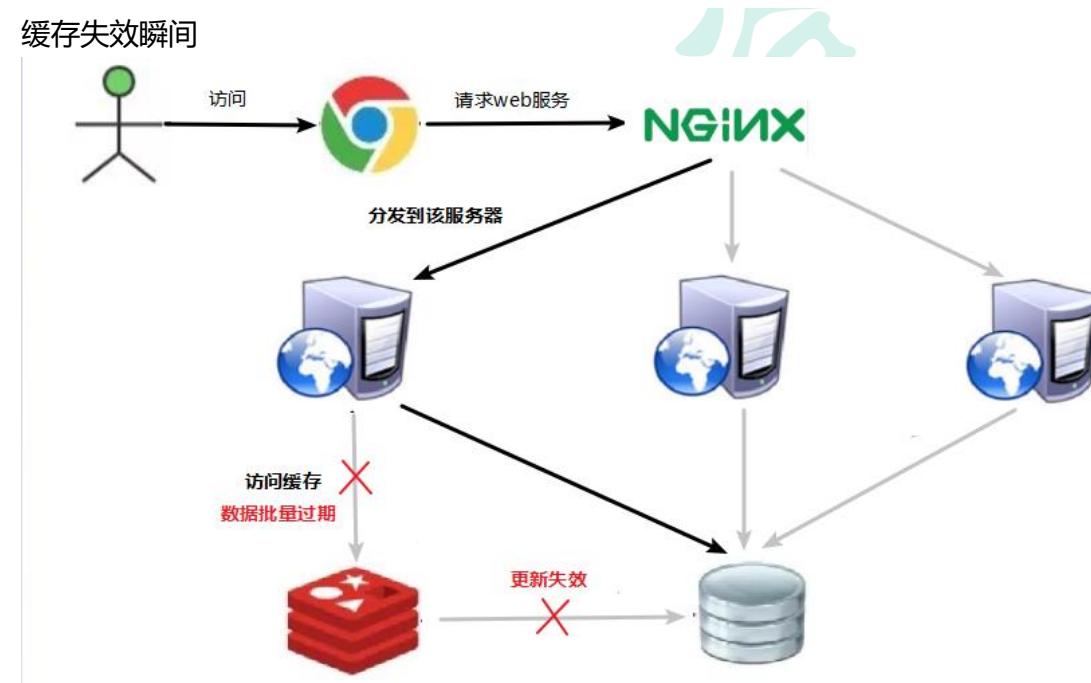
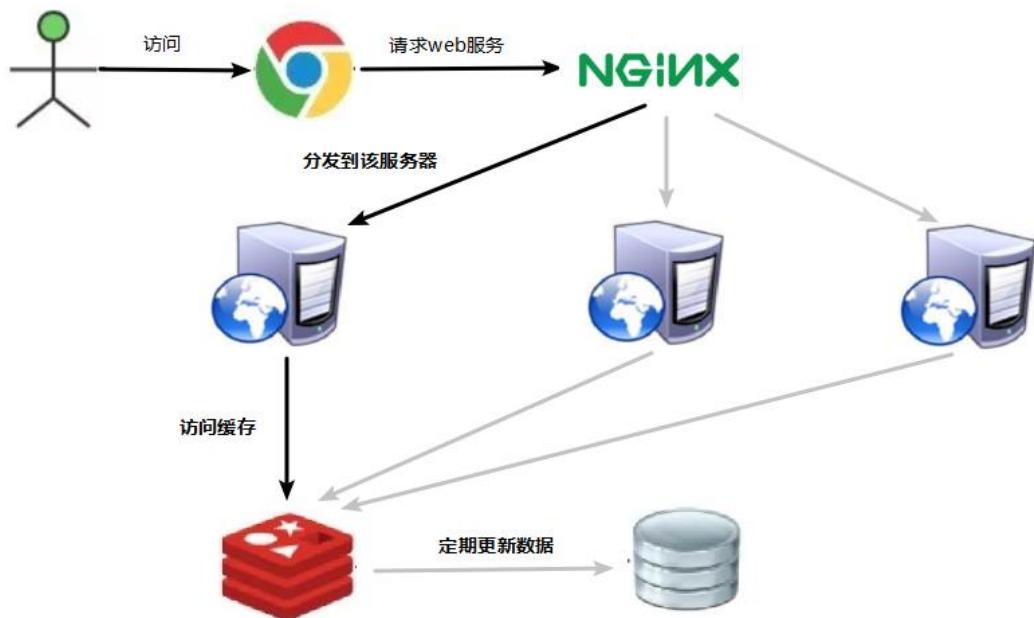
## 16.3. 缓存雪崩

### 16.3.1. 问题描述

key 对应的数据存在，但在 redis 中过期，此时若有大量并发请求过来，这些请求发现缓存过期一般都会从后端 DB 加载数据并回设到缓存，这个时候大并发的请求可能会瞬间把后端 DB 压垮。

缓存雪崩与缓存击穿的区别在于这里针对很多 key 缓存，前者则是某一个 key

正常访问



### 16.3.2. 解决方案

缓存失效时的雪崩效应对底层系统的冲击非常可怕!

解决方案：

(1) 构建多级缓存架构：nginx 缓存 + redis 缓存 + 其他缓存 (ehcache 等)

(2) 使用锁或队列：

用加锁或者队列的方式保证来保证不会有大量的线程对数据库一次性进行读写，从而避免失效时大量的并发请求落到底层存储系统上。不适用高并发情况

**(3) 设置过期标志更新缓存:**

记录缓存数据是否过期（设置提前量），如果过期会触发通知另外的线程在后台去更新实际 key 的缓存。

**(4) 将缓存失效时间分散开:**

比如我们可以在原有的失效时间基础上增加一个随机值，比如 1-5 分钟随机，这样每一个缓存的过期时间的重复率就会降低，就很难引发集体失效的事件。

## 16.4. 分布式锁

### 16.4.1. 问题描述

随着业务发展的需要，原单体单机部署的系统被演化成分布式集群系统后，由于分布式系统多线程、多进程并且分布在不同机器上，这将使原单机部署情况下的并发控制锁策略失效，单纯的 Java API 并不能提供分布式锁的能力。为了解决这个问题就需要一种跨 JVM 的互斥机制来控制共享资源的访问，这就是分布式锁要解决的问题！

分布式锁主流的实现方案：

1. 基于数据库实现分布式锁
2. 基于缓存（Redis 等）
3. 基于 Zookeeper

每一种分布式锁解决方案都有各自的优缺点：

1. 性能：redis 最高
2. 可靠性：zookeeper 最高

这里，我们就基于 redis 实现分布式锁。

### 16.4.2. 解决方案：使用 redis 实现分布式锁

redis:命令

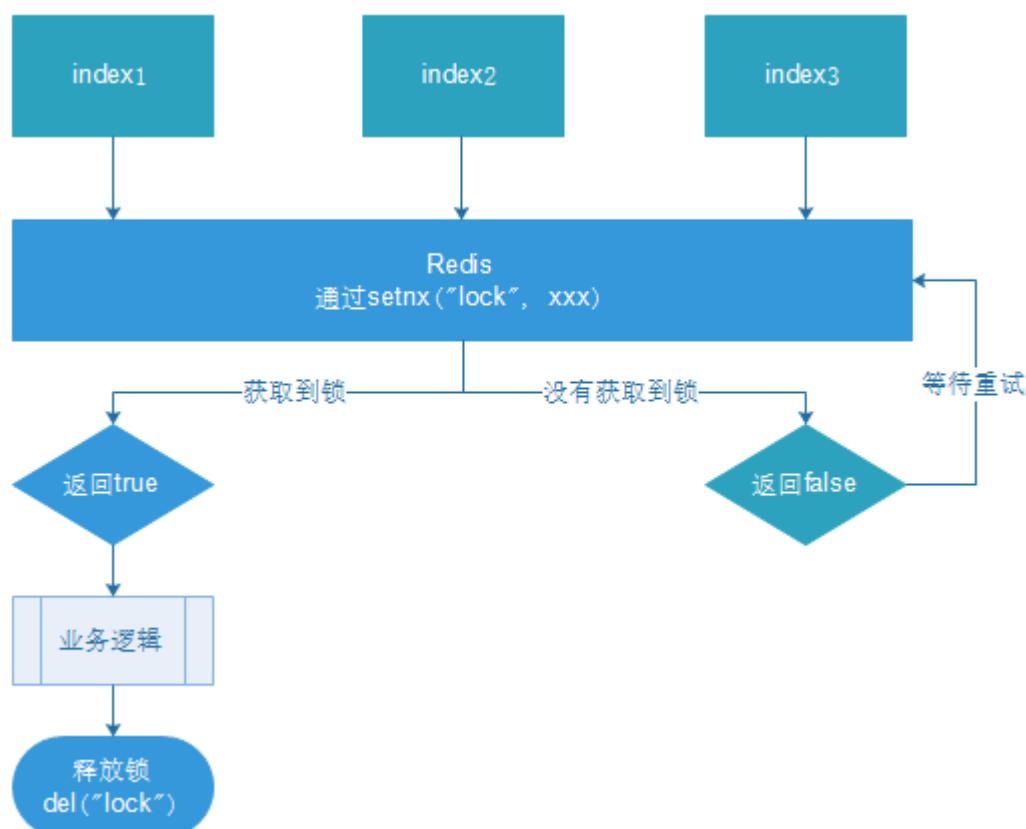
```
# set sku:1:info "OK" NX PX 10000
```

EX second：设置键的过期时间为 second 秒。SET key value EX second 效果等同于SETEX key second value。

PX millisecond : 设置键的过期时间为 millisecond 毫秒。 SET key value PX millisecond 效果等同于 PSETEX key millisecond value 。

NX : 只在键不存在时，才对键进行设置操作。 SET key value NX 效果等同于 SETNX key value 。

XX : 只在键已经存在时，才对键进行设置操作。



1. 多个客户端同时获取锁 (`setnx`)
2. 获取成功，执行业务逻辑{从 db 获取数据，放入缓存}，执行完成释放锁 (`del`)
3. 其他客户端等待重试

### 16.4.3. 编写代码

Redis: set num 0

```
@GetMapping("testLock")
public void testLock() {
    //1 获取锁, setne
    Boolean lock = redisTemplate.opsForValue().setIfAbsent("lock", "111");
    //2 获取锁成功、查询num的值
```

```
if(lock) {  
    Object value = redisTemplate.opsForValue().get("num");  
    //2.1 判断 num 为空 return  
    if(StringUtils.isEmpty(value)) {  
        return;  
    }  
    //2.2 有值就转成成 int  
    int num = Integer.parseInt(value+"");  
    //2.3 把 redis 的 num 加 1  
    redisTemplate.opsForValue().set("num", ++num);  
    //2.4 释放锁, del  
    redisTemplate.delete("lock");  
  
} else {  
    //3 获取锁失败、每隔 0.1 秒再获取  
    try {  
        Thread.sleep(100);  
        testLock();  
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }  
}  
}
```

重启，服务集群，通过网关压力测试：

```
ab -n 1000 -c 100 http://192.168.140.1:8080/test/testLock
```

```
[root@cocoon ~]# ab -n 5000 -c 100 http://192.168.140.1:8080/test/testLock  
This is ApacheBench, Version 2.3 <$Revision: 1430300 $>  
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/  
Licensed to The Apache Software Foundation, http://www.apache.org/  
  
Benchmarking 192.168.140.1 (be patient)  
Completed 500 requests  
Completed 1000 requests  
Completed 1500 requests  
Completed 2000 requests  
Completed 2500 requests  
Completed 3000 requests  
Completed 3500 requests  
Completed 4000 requests  
Completed 4500 requests  
Completed 5000 requests  
Finished 5000 requests
```

查看 redis 中 num 的值：

```
127.0.0.1:6379> get num  
"5000"  
127.0.0.1:6379>
```

基本实现。

问题：`setnx` 刚好获取到锁，业务逻辑出现异常，导致锁无法释放

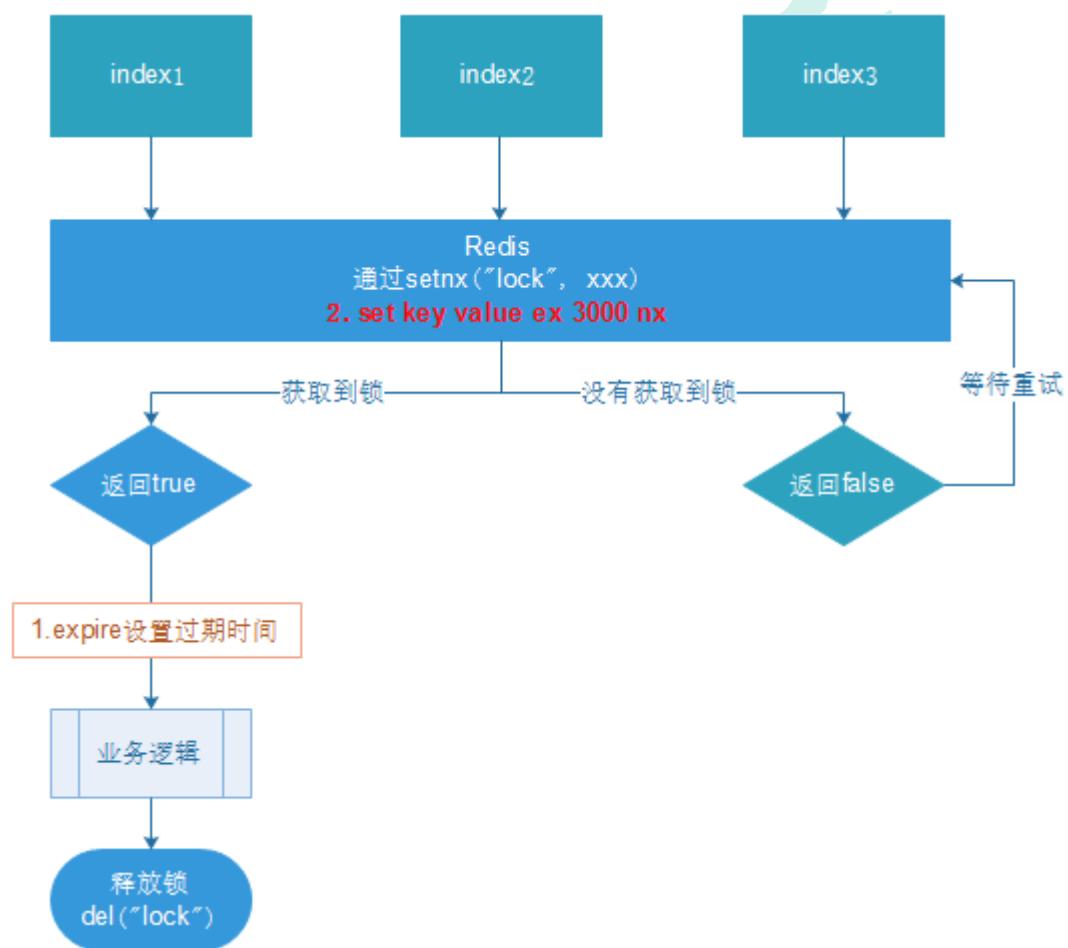
解决：设置过期时间，自动释放锁。

#### 16.4.4. 优化之设置锁的过期时间

设置过期时间有两种方式：

1. 首先想到通过 `expire` 设置过期时间（缺乏原子性：如果在 `setnx` 和 `expire` 之间出现异常，锁也无法释放）

2. 在 `set` 时指定过期时间（推荐）



设置过期时间：

```
54
55 @Override
56 public void testLock() {
57     // 1. 从redis中获取锁, setnx
58     Boolean lock = this.redisTemplate.opsForValue()
59         .setIfAbsent("lock", "111", timeout: 3, TimeUnit.SECONDS);
60     if (lock) {
61         // 查询redis中的num值
62         String value = this.redisTemplate.opsForValue().get("num");
63         // 没有该值return
64         if (StringUtils.isBlank(value)) {
65             return;
66         }
67         // 有值就转成int
```

压力测试肯定也没有问题。自行测试

问题：可能会释放其他服务器的锁。

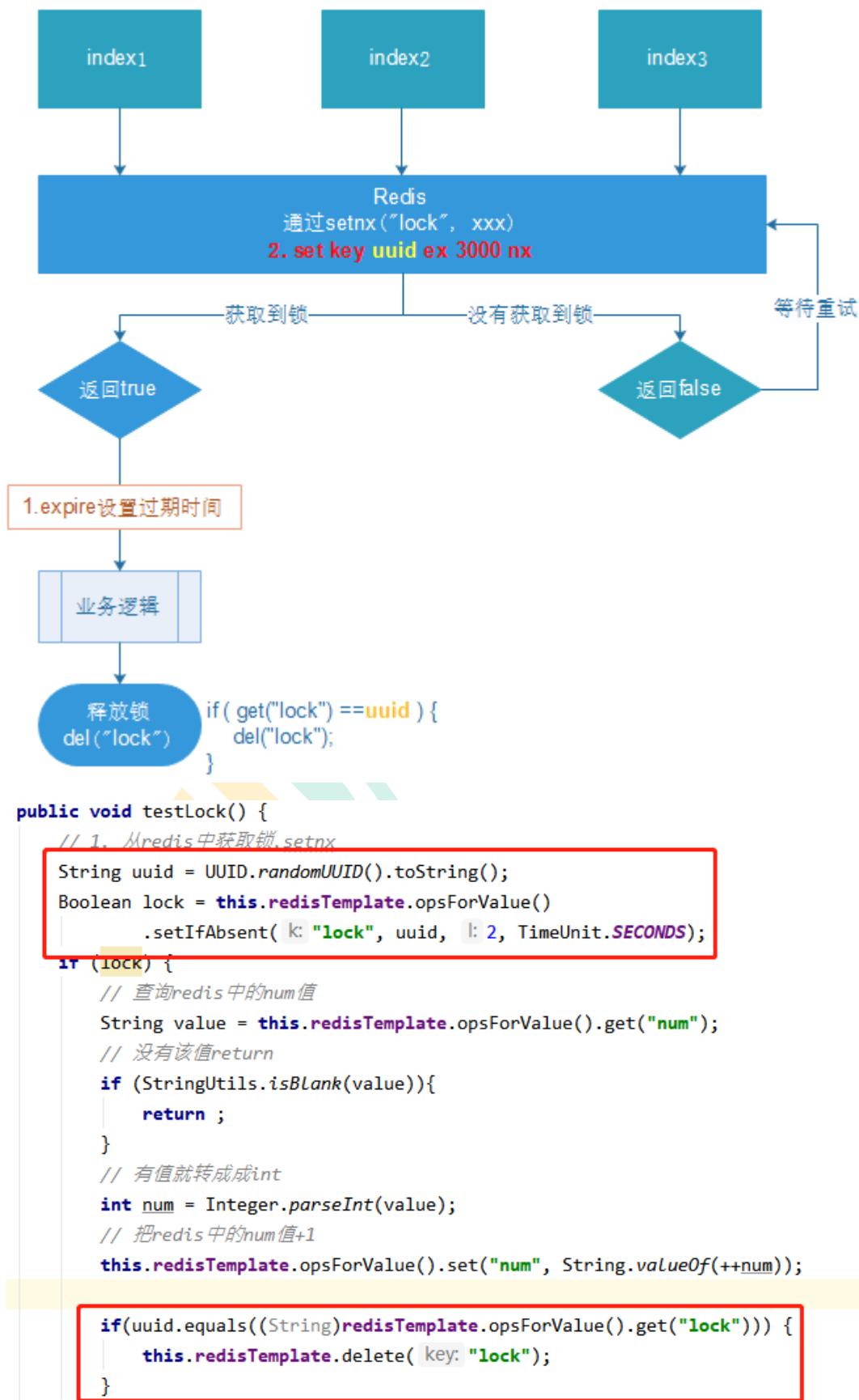
场景：如果业务逻辑的执行时间是 7s。执行流程如下

1. index1 业务逻辑没执行完，3秒后锁被自动释放。
2. index2 获取到锁，执行业务逻辑，3秒后锁被自动释放。
3. index3 获取到锁，执行业务逻辑
4. index1 业务逻辑执行完成，开始调用 del 释放锁，这时释放的是 index3 的锁，导致 index3 的业务只执行 1s 就被别人释放。

最终等于没锁的情况。

解决：setnx 获取锁时，设置一个指定的唯一值（例如：uuid）；释放前获取这个值，判断是否自己的锁

### 16.4.5. 优化之 UUID 防误删



问题：删除操作缺乏原子性。

场景：

1. index1 执行删除时，查询到的 lock 值确实和 uuid 相等

```
uuid=v1  
set(lock,uuid);  
if(uuid.equals((String)redisTemplate.opsForValue().get("lock")))
```

2. index1 执行删除前，lock 刚好过期时间已到，被 redis 自动释放

在 redis 中没有了 lock，没有了锁。

```
this.redisTemplate.delete( key: "lock");
```

3. index2 获取了 lock

index2 线程获取到了 cpu 的资源，开始执行方法

```
uuid=v2  
set(lock,uuid);
```

4. index1 执行删除，此时会把 index2 的 lock 删除

index1 因为已经在方法中了，所以不需要重新上锁。index1 有执行的权限。index1 已经比较完成了，这个时候，开始执行

```
this.redisTemplate.delete( key: "lock");
```

删除的 index2 的锁！

#### 16.4.6. 优化之 LUA 脚本保证删除的原子性

```
@GetMapping("testLockLua")  
public void testLockLua() {  
    //1 声明一个 uuid，将做为一个 value 放入我们的 key 所对应的值中  
    String uuid = UUID.randomUUID().toString();  
    //2 定义一个锁：lua 脚本可以使用同一把锁，来实现删除！  
    String skuId = "25"; // 访问 skuId 为 25 号的商品 100008348542  
    String locKey = "lock:" + skuId; // 锁住的是每个商品的数据
```

```
// 3 获取锁
Boolean lock = redisTemplate.opsForValue().setIfAbsent(lockKey, uuid, 3,
TimeUnit.SECONDS);

// 第一种： lock 与过期时间中间不写任何的代码。
// redisTemplate.expire("lock", 10, TimeUnit.SECONDS); //设置过期时间
// 如果 true
if (lock) {
    // 执行的业务逻辑开始
    // 获取缓存中的 num 数据
    Object value = redisTemplate.opsForValue().get("num");
    // 如果是空直接返回
    if (StringUtils.isEmpty(value)) {
        return;
    }
    // 不是空 如果说在这出现了异常！ 那么 delete 就删除失败！ 也就是说锁永远存在！
    int num = Integer.parseInt(value + "");
    // 使 num 每次+1 放入缓存
    redisTemplate.opsForValue().set("num", String.valueOf(++num));
    /*使用 lua 脚本来锁*/
    // 定义 lua 脚本
    String script = "if redis.call('get', KEYS[1]) == ARGV[1] then return
redis.call('del', KEYS[1]) else return 0 end";
    // 使用 redis 执行 lua 执行
    DefaultRedisScript<Long> redisScript = new DefaultRedisScript<>();
    redisScript.setScriptText(script);
    // 设置一下返回值类型 为 Long
    // 因为删除判断的时候，返回的 0，给其封装为数据类型。如果不封装那么默认
    // 返回 String 类型，
    // 那么返回字符串与 0 会有发生错误。
    redisScript.setResultType(Long.class);
    // 第一个要是 script 脚本，第二个需要判断的 key，第三个就是 key 所对应的
    // 值。
    redisTemplate.execute(redisScript, Arrays.asList(lockKey), uuid);
} else {
    // 其他线程等待
    try {
        // 睡眠
        Thread.sleep(1000);
        // 睡醒了之后，调用方法。
        testLockLua();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

Lua 脚本详解：

客户端执行以上的命令：

- 如果服务器返回 OK，那么这个客户端获得锁。
- 如果服务器返回 NIL，那么客户端获取锁失败，可以在稍后再重试。

设置的过期时间到达之后，锁将自动释放。

可以通过以下修改，让这个锁实现更健壮：

- 不使用固定的字符串作为键的值，而是设置一个不可猜测（non-guessable）的长随机字符串，作为口令串（token）。
- 不使用 DEL 命令来释放锁，而是发送一个 Lua 脚本，这个脚本只在客户端传入的值和键的口令串相匹配时，才对键进行删除。

这两个改动可以防止持有过期锁的客户端误删现有锁的情况出现。

以下是一个简单的解锁脚本示例：

```
if redis.call("get",KEYS[1]) == ARGV[1]
then
    return redis.call("del",KEYS[1])
else
    return 0
end
```

## 项目中正确使用：

1. 定义 key，key 应该是为每个 sku 定义的，也就是每个 sku 有一把锁。

```
String lockKey = "lock:" + skuId; // 锁住的是每个商品的数据

Boolean lock = redisTemplate.opsForValue().setIfAbsent(lockKey,
uuid, 3, TimeUnit.SECONDS);
```

```
/* 使用Lua脚本来锁*/
// 定义Lua脚本
String script="if redis.call('get', KEYS[1]) == ARGV[1] then return redis.call('del', KEYS[1]) else return 0 end";
// 使用redisTemplate执行Lua执行
// 第一种传值
// DefaultRedisScript<Object> redisScript = new DefaultRedisScript<>(script);
DefaultRedisScript<Long> redisScript = new DefaultRedisScript<>();
// 第二种传值
redisScript.setScriptText(script);
// 设置一下返回值类型为Long
// 因为删除判断的时候，返回的0，将其封装为数据类型。如果不封装那么默认返回String类型，那么返回字符串与0会有发生错误。
redisScript.setResultType(Long.class);
// 第一个要是script脚本，第二个需要判断的key，第三个就是key所对应的值。
redisTemplate.execute(redisScript, Arrays.asList(lockKey), uuid);
```

## 16.4.7. 总结

### 1、加锁

```
// 1. 从redis中获取锁, set k1 v1 px 20000 nx
String uuid = UUID.randomUUID().toString();
Boolean lock = this.redisTemplate.opsForValue()
.setIfAbsent("lock", uuid, 2, TimeUnit.SECONDS);
```

### 2、使用lua释放锁

```
// 2. 释放锁 del
String script = "if redis.call('get', KEYS[1]) == ARGV[1] then return
redis.call('del', KEYS[1]) else return 0 end";
// 设置 Lua 脚本返回的数据类型
DefaultRedisScript<Long> redisScript = new DefaultRedisScript<>();
// 设置 Lua 脚本返回类型为 Long
redisScript.setResultType(Long.class);
redisScript.setScriptText(script);
redisTemplate.execute(redisScript, Arrays.asList("lock"),uuid);
```

### 3、重试

```
Thread.sleep(500);
testLock();
```

为了确保分布式锁可用，我们至少要确保锁的实现同时满足以下四个条件：

- 互斥性。在任意时刻，只有一个客户端能持有锁。
- 不会发生死锁。即使有一个客户端在持有锁的期间崩溃而没有主动解锁，也能保证后续其他客户端能加锁。
- 解铃还须系铃人。加锁和解锁必须是同一个客户端，客户端自己不能把别人加的锁给解了。
- 加锁和解锁必须具有原子性。

## 17. Redis 6.0 新功能

### 17.1. ACL

#### 17.1.1. 简介

Redis ACL 是 Access Control List (访问控制列表) 的缩写，该功能允许根据可以执行的命令和可以访问的键来限制某些连接。

在 Redis 5 版本之前，Redis 安全规则只有密码控制 还有通过 rename 来调整高危命令比如 flushdb , KEYS\* , shutdown 等。Redis 6 则提供 ACL 的功能对用户进行更细粒度的权限控制：

- (1) 接入权限:用户名和密码
- (2) 可以执行的命令
- (3) 可以操作的 KEY

参考官网: <https://redis.io/topics/acl>

### 17.1.2. 命令

1、使用 acl list 命令展现用户权限列表

(1) 数据说明

```
127.0.0.1:6379> acl list
1) "user default on nopass ~* +@all"
```

用户名: user  
是否启用 (on/off): default  
密码 (没密码): nopass  
可操作的key: ~\*  
可执行的命令: +@all

2、使用 acl cat 命令

(1) 查看添加权限指令类别

```
127.0.0.1:6379> acl cat
1) "keyspace"
2) "read"
3) "write"
4) "set"
5) "sortedset"
6) "list"
7) "hash"
8) "string"
9) "bitmap"
10) "hyperloglog"
11) "geo"
12) "stream"
13) "pubsub"
14) "admin"
15) "fast"
16) "slow"
17) "blocking"
18) "dangerous"
19) "connection"
20) "transaction"
21) "scripting"
```

(2) 加参数类型名可以查看类型下具体命令

```
127.0.0.1:6379> acl cat string
1) "setrange"
2) "decrby"
3) "substr"
4) "stralgo"
5) "decr"
6) "incrby"
7) "strlen"
8) "msetnx"
9) "get"
10) "incr"
11) "setex"
12) "setnx"
13) "incrbyfloat"
14) "mset"
15) "append"
16) "getrange"
17) "mget"
18) "psetex"
19) "set"
20) "getset"
```

### 3、使用 acl whoami 命令查看当前用户

```
127.0.0.1:6379> acl whoami
"default"
```

### 4、使用 aclsetuser 命令创建和编辑用户 ACL

#### (1) ACL 规则

下面是有效 ACL 规则的列表。某些规则只是用于激活或删除标志，或对用户 ACL 执行给定更改的单个单词。其他规则是字符前缀，它们与命令或类别名称、键模式等连接在一起。

ACL 规则		
类型	参数	说明
启动和禁用用户	on	激活某用户账号
	off	禁用某用户账号。注意，已验证的连接仍然可以工作。如果默认用户被标记为 off，则新连接将在未进行身份验证的情况下启动，并要求用户使用 AUTH 选项发送 AUTH 或 HELLO，以便以某种方式进行身份验证。
权限的添加删除	+<command>	将指令添加到用户可以调用的指令列表中
	-<command>	从用户可执行指令列表移除指令
@<category>	+@<category>	添加该类别中用户要调用的所有指令，有效类别为 @admin、@set、@sortedset 等，通过调用 ACL CAT 命令查看完整列表。特殊类别@all 表示所有命令，包括当前存在于服务器中的命令，以及将来将通过模块加载的命令。
	-@<category>	从用户可调用指令中移除类别

	<b>allcommands</b>	+@all 的别名
	<b>nocommand</b>	-@all 的别名
可操作键的添加或删除	<b>~&lt;pattern&gt;</b>	添加可作为用户可操作的键的模式。例如~*允许所有的键

### (2) 通过命令创建新用户默认权限

acl setuser user1

```
127.0.0.1:6379> acl setuser user1
OK
127.0.0.1:6379> acl list
1) "user default on nopass ~* +@all"
2) "user user1 off -@all"
```

在上面的示例中，我根本没有指定任何规则。如果用户不存在，这将使用 just created 的默认属性来创建用户。如果用户已经存在，则上面的命令将不执行任何操作。

### (3) 设置有用户名、密码、ACL 权限、并启用的用户

acl setuser user2 on >password ~cached:\* +get

```
127.0.0.1:6379> acl setuser user2 on >password ~cached:* +get
OK
127.0.0.1:6379> acl list
1) "user default on nopass ~* +@all"
2) "user user1 off -@all"
3) "user user2 on #5e884898da28047151d0e56f8dc6292773603d0d6aabdd62a11ef721d1542d8
   ~cached:* -@all +get"
127.0.0.1:6379>
```

### (4) 切换用户，验证权限

```
127.0.0.1:6379> acl whoami
"default"
127.0.0.1:6379> auth user2 password
OK
127.0.0.1:6379> acl whoami
(error) NOPERM this user has no permissions to run the 'acl' command or its subcommands
127.0.0.1:6379> get foo
(error) NOPERM this user has no permissions to access one of the keys used as arguments
127.0.0.1:6379> get cached:1121
(nil)
127.0.0.1:6379> set cached:1121 1121
(error) NOPERM this user has no permissions to run the 'set' command or its subcommands
```

## 17.2. IO 多线程

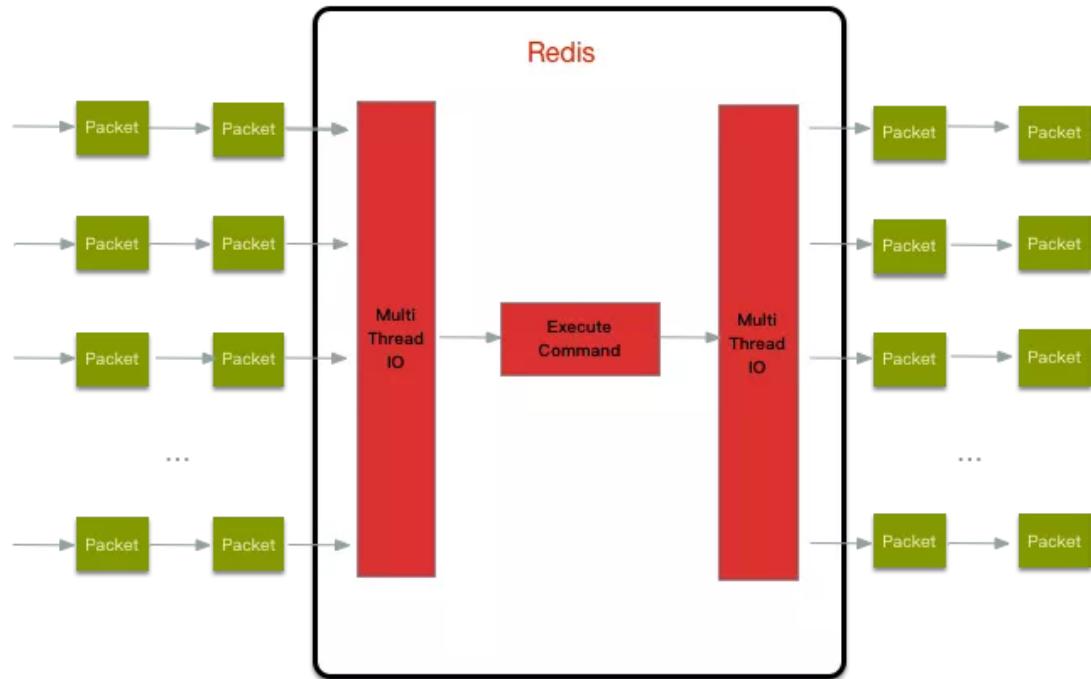
### 17.2.1. 简介

Redis6 终于支撑多线程了，告别单线程了吗？

IO 多线程其实指客户端交互部分的网络 IO 交互处理模块多线程，而非执行命令多线程。Redis6 执行命令依然是单线程。

### 17.2.2. 原理架构

Redis 6 加入多线程, 但跟 Memcached 这种从 IO 处理到数据访问多线程的实现模式有些差异。Redis 的多线程部分只是用来处理网络数据的读写和协议解析, 执行命令仍然是单线程。之所以这么设计是不想因为多线程而变得复杂, 需要去控制 key、lua、事务, LPUSH/LPOP 等等的并发问题。整体的设计大体如下:



另外, 多线程 IO 默认也是不开启的, 需要在配置文件中配置

io-threads-do-reads yes

io-threads 4

### 17.3. 工具支持 Cluster

之前老版 Redis 想要搭集群需要单独安装 ruby 环境, Redis 5 将 redis-trib.rb 的功能集成到 redis-cli。另外官方 redis-benchmark 工具开始支持 cluster 模式了, 通过多线程的方式对多个分片进行压测。

```
[root@cocoon bin]# redis-benchmark --help
Usage: redis-benchmark [-h <host>] [-p <port>] [-c <clients>] [-n <requests>] [-k <bo
-h <hostname>      Server hostname (default 127.0.0.1)
-p <port>          Server port (default 6379)
-s <socket>        Server socket (overrides host and port)
-a <password>     Password for Redis Auth
--user <username> Used to send ACL style 'AUTH username pass'. Needs -a.
-c <clients>       Number of parallel connections (default 50)
-n <requests>     Total number of requests (default 100000)
-d <size>          Data size of SET/GET value in bytes (default 3)
--dbnum <db>       SELECT the specified db number (default 0)
--threads <num>   Enable multi-thread mode.
--cluster          Enable cluster mode.
--enable-tracking Send CLIENT TRACKING on before starting benchmark.
-k <boolean>      1=keep alive 0=reconnect (default 1)
-r <keyspacelen>  Use random keys for SET/GET/INCR, random values for SADD,
random members and scores for ZADD.
```

## 17.4. Redis 新功能持续关注

Redis6 新功能还有：

- 1、RESP3 新的 Redis 通信协议：优化服务端与客户端之间通信
- 2、Client side caching 客户端缓存：基于 RESP3 协议实现的客户端缓存功能。为了进一步提升缓存的性能，将客户端经常访问的数据 cache 到客户端。减少 TCP 网络交互。
- 3、Proxy 集群代理模式：Proxy 功能，让 Cluster 拥有像单实例一样的接入方式，降低大家使用 cluster 的门槛。不过需要注意的是代理不改变 Cluster 的功能限制，不支持的命令还是不会支持，比如跨 slot 的多 Key 操作。
- 4、Modules API

Redis 6 中模块 API 开发进展非常大，因为 Redis Labs 为了开发复杂的功能，从一开始就用上 Redis 模块。Redis 可以变成一个框架，利用 Modules 来构建不同系统，而不需要从头开始写然后还要 BSD 许可。Redis 一开始就是一个向编写各种系统开放的平台。