

```
[root@localhost redis6380]# kill -9 2910
```

```
[root@localhost redis6380]# ps -aux |grep redis
Warning: bad syntax, perhaps a bogus '-'? See /usr/share/doc/procps-3.2.8/FAQ
root      2910  0.1  1.2 167776 12148 ?        Ssl  18:41   0:07 ./bin/redis-server 127.0.0.1:6380
root      2916  0.1  1.2 161632 12312 ?        Ssl  18:41   0:07 ./bin/redis-server 127.0.0.1:6381
root      2990  0.3  0.8 152416  8244 pts/4    Sl   18:53   0:10 ./redis-sentinel *:26379 [sentinel]
root      3019  0.0  0.7  23864  7324 pts/3    S+   18:57   0:00 ./bin/redis-cli -h 192.168.197.129 -p 6379
root      3029  0.0  0.7  23996  7504 pts/1    S+   18:59   0:00 ./bin/redis-cli -h 127.0.0.1 -p 6380
root      3030  0.0  0.7  23996  7504 pts/2    S+   19:00   0:00 ./bin/redis-cli -h 127.0.0.1 -p 6381
root      3342  0.1  1.6 167776 16248 ?        Ssl  19:25   0:01 ./bin/redis-server 192.168.197.129:6379
```

1.了解NoSql

1.1 什么是Nosql

NoSQL，泛指非关系型的数据库。随着互联网web2.0网站的兴起，传统的关系数据库在处理web2.0网站，特别是超大规模和高并发的SNS类型的web2.0纯动态网站已经显得力不从心，出现了很多难以克服的问题，而非关系型的数据库则由于其本身的特点得到了非常迅速的发展。NoSQL数据库的产生就是为了解决大规模数据集合，多重数据种类带来的挑战，尤其是大数据应用难题。

NoSQL最常见的解释是“non-relational”，“Not Only SQL”也被很多人接受。NoSQL仅仅是一个概念，泛指非关系型的数据库，区别于关系数据库，它们不保证关系数据的ACID特性。

1.2 为什么要使用NoSql

传统的数据库遇到的瓶颈

传统的关系数据库具有不错的性能，高稳定型，久经历史考验，而且使用简单，功能强大，同时也积累了大量的成功案例。在互联网领域，MySQL成为了绝对靠前的王者，毫不夸张的说，MySQL为互联网的发展做出了卓越的贡献。

在90年代，一个网站的访问量一般都不大，用单个数据库完全可以轻松应付。在那个时候，更多的都是静态网页，动态交互类型的网站不多。

到了最近10年，网站开始快速发展。火爆的论坛、博客、sns、微博逐渐引领web领域的潮流。在初期，论坛的流量其实也不大，如果你接触网络比较早，你可能还记得那个时候还有文本型存储的论坛程序，可以想象一般的论坛的流量有多大。

现在网站的特点:

(1) 高并发读写

Web2.0网站，数据库并发负载非常高，往往达到每秒上万次的读写请求

(2) 高容量存储和高效存储

Web2.0网站通常需要在后台数据库中存储海量数据，如何存储海量数据并进行高效的查询往往是一个挑战

(3) 高扩展性和高可用性

随着系统的用户量和访问量与日俱增，需要数据库能够很方便的进行扩展、维护

1.3 NoSql数据库的优势

(1) 易扩展

NoSQL数据库种类繁多，但是一个共同的特点都是去掉关系数据库的关系型特性。**数据之间无关系**，这样就非常容易扩展。也无形之间，在架构的层面上带来了可扩展的能力。

(2)大数据量，高性能

NoSQL数据库都具有非常高的读写性能，尤其在大数据量下，同样表现优秀。这得益于它的无关系性，数据库的结构简单。一般MySQL使用Query Cache，每次表的更新Cache就失效，是一种大粒度的Cache，在针对web2.0的交互频繁的应用，Cache性能不高。而NoSQL的Cache是记录级的，是一种细粒度的Cache，所以NoSQL在这个层面上来说就要性能高很多了。

(3)灵活的数据模型

NoSQL无需事先为要存储的数据建立字段，随时可以存储自定义的数据格式。而在关系数据库里，增删字段是一件非常麻烦的事情。如果是非常大数据量的表，增加字段简直就是一个噩梦。这点在大数据量的web2.0时代尤其明显。

(4) 高可用

NoSQL在不太影响性能的情况，就可以方便的实现高可用的架构。比如Cassandra，HBase模型，通过复制模型也能实现高可用。

1.4 常见的NoSql产品



1.5 各产品的区别

NoSQL 数据库分类和特点					
分类	相关产品	应用场景	数据模型	优点	缺点
键值数据库	Redis、Memcached、Riak	内容缓存，如会话、配置文件、参数等；频繁读写、拥有简单数据模型的应用	<key,value> 键值对，通过散列表来实现	扩展性好，灵活性好，大量操作时性能高	数据无结构化，通常只被当做字符串或者二进制数据，只能通过键来查询值
列族数据库	Bigtable、HBase、Cassandra	分布式数据存储与管理	以列族式存储，将同一列数据存在一起	可扩展性强，查找速度快，复杂性低	功能局限，不支持事务的强一致性
文档数据库	MongoDB、CouchDB	Web 应用，存储面向文档或类似半结构化的数据	<key,value> value 是 JSON 结构的文档	数据结构灵活，可以根据value 构建索引	缺乏统一查询语法
图形数据库	Neo4j、InfoGrid	社交网络、推荐系统，专注构建关系图谱	图结构	支持复杂的图形算法	复杂性高，只能支持一定的数据规模

2.Redis介绍

2.1什么是Redis

全称：REmote DIctionary Server（远程字典服务器）。是完全开源免费的，用C语言编写的，遵守BCD协议。是一个高性能的(key/value)分布式内存数据库，

基于内存运行并支持持久化的NoSQL数据库，是当前最热门的NoSql数据库之一,也被人们称为数据结构服务器。

Redis 与其他 key - value 缓存产品有以下三个特点

- (1) Redis支持数据的持久化，可以将内存中的数据保持在磁盘中，重启的时候可以再次加载进行使用
- (2) Redis不仅仅支持简单的key-value类型的数据，同时还提供list，set，zset，hash等数据结构的存储

(3) Redis支持数据的备份，即master-slave(主从)模式的数据备份

2.2 Redis优势

(1) 性能极高 – Redis能读的速度是110000次/s,写的速度是81000次/s。

(2) 丰富的数据类型 – Redis支持二进制案例的 Strings, Lists, Hashes, Sets 及 Ordered Sets 数据类型操作。

(3) 原子 – Redis的所有操作都是原子性的，同时Redis还支持对几个操作全并后的原子性执行。

(4) 丰富的特性 – Redis还支持 publish/subscribe, 通知, key 过期等等特性

(5) 采用**单线程**，避免了不必要的上下文切换和竞争条件，也不存在多进程或者多线程导致的切换而消耗 CPU，不用去考虑各种锁的问题，不存在加锁释放锁操作，没有因为可能出现死锁而导致的性能消耗；

(6) 使用多路I/O复用模型，非阻塞IO；

2.3 Redis应用场景

(1) 缓存(数据查询，短连接，新闻内容，商品内容等)，使用最多

(2) 聊天室在线好友列表

(3) 任务队列(秒杀，抢购，12306等)

(4) 应用排行榜

(5) 网站访问统计

(6) 数据过期处理(可以精确到毫秒)

(7) 分布式集群架构中的session问题

2.4 Redis下载

(1) [Http://redis.io/](http://redis.io/) 英文地址

(2) [Http://www.redis.cn/](http://www.redis.cn/) 中文地址

3.Linux下安装Redis

3.1 环境准备

(1)虚拟机版本:VMware® Workstation 12 Pro

(2) Linux系统:Centos Release 6.5

(3) 远程命令端:xshell

(4)文件传输工具:SecureFXPortable

3.2 Redis的安装

3.2.1 Redis的编译环境

Redis是C语言开发的，安装redis需要先去官网下载源码进行编译，编译需要依赖于GCC编译环境，如果CentOS上没有安装gcc编译环境，需要提前安装，安装命令如下：(这里我们使用root用户处理这些操作)

```
[root@localhost ~]# yum install gcc-c++
```

如果提示是否下载，选择: y

```
Transaction Summary
=====
Install      2 Package(s)
Upgrade     5 Package(s)

Total download size: 21 M
Is this ok [y/N]: y
```

如果提示是否安装,选择: y

```
Dependency Installed:
  libstdc++-devel.x86_64 0:4.4.7-23.el6

Dependency Updated:
  cpp.x86_64 0:4.4.7-23.el6
  libstdc++.x86_64 0:4.4.7-23.el6

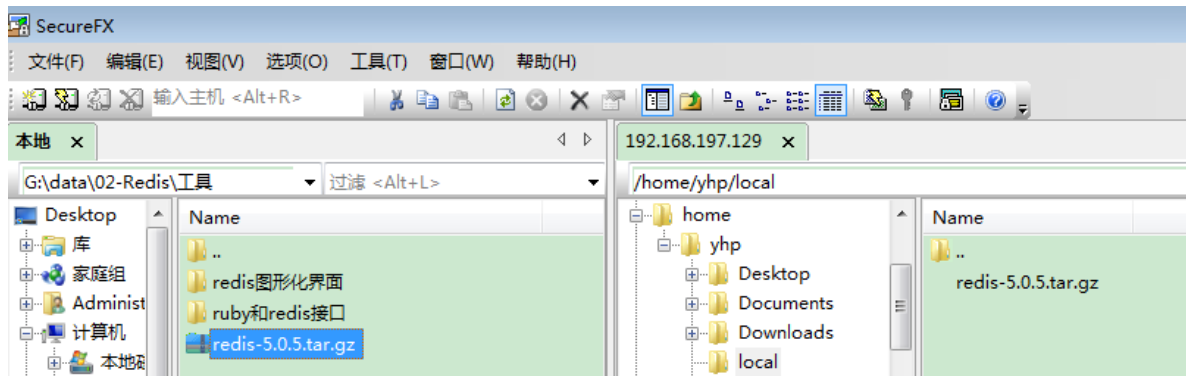
Complete!
[root@localhost ~]#
```

3.2.2 Redis的安装

(1) 使用SecureFXPortable上传Redis安装文件到Linux目录



(2) 上传Redis安装文件,这里我上传自建文件夹: /home/yhp/local



(3)解压redis文件

```
[root@localhost local]# tar -zxvf redis-5.0.5.tar.gz
```

```
drwxrwxr-x. 6 root root    4096 May 15  2019 redis-5.0.5
-rw-rw-r--. 1 yhp  yhp  1975750 May 28 03:02 redis-5.0.5.tar.gz
```

(4)编译Redis(编译,将.c文件编译为.o文件)

进入解压文件夹,执行 make

```
[root@localhost local]# cd redis-5.0.5
[root@localhost redis-5.0.5]# make
```

```
make[1]: Leaving directory `/home/yhp/local/redis-5.0.5/src'
[root@localhost redis-5.0.5]#
```

编译成功! 如果编译过程中出错, 先删除安装文件目录, 后解压重新编译。

(5) 安装

```
[root@localhost redis-5.0.5]# make PREFIX=/home/admin/myapps/redis install
```

说明:这里的/home/myapps/redis 是自定义的redis安装路径

```
[root@localhost redis-5.0.5]# make PREFIX=/home/admin/myapps/redis install
cd src && make install
make[1]: Entering directory `/home/yhp/local/redis-5.0.5/src'
  CC Makefile.dep
make[1]: Leaving directory `/home/yhp/local/redis-5.0.5/src'
make[1]: Entering directory `/home/yhp/local/redis-5.0.5/src'

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

INSTALL install
INSTALL install
INSTALL install
INSTALL install
INSTALL install
make[1]: Leaving directory `/home/yhp/local/redis-5.0.5/src'
```

(6)安装之后的bin目录

```
[root@localhost redis]# cd /home/admin/myapps/redis/bin
[root@localhost bin]#
```

bin文件夹下的命令:

```
-rwxr-xr-x. 1 root root 9200267 May 29 10:14 redis-benchmark
-rwxr-xr-x. 1 root root 12274509 May 29 10:14 redis-check-aof
-rwxr-xr-x. 1 root root 12274509 May 29 10:14 redis-check-rdb
-rwxr-xr-x. 1 root root 9577312 May 29 10:14 redis-cli
lrwxrwxrwx. 1 root root 12 May 29 10:14 redis-sentinel -> redis-server
-rwxr-xr-x. 1 root root 12274509 May 29 10:14 redis-server
```

(7) Copy文件

Redis启动需要一个配置文件，可以修改端口号信息。将redis解压的文件夹中的redis.conf文件复制到安装目录

```
[root@localhost redis-5.0.5]# cp redis.conf /home/admin/myapps/redis
```

3.3 Redis的启动

3.3.1 Redis的前端模式启动

直接运行bin/redis-server将使永前端模式启动，前端模式启动的缺点是启动完成后，不能再进行其他操作，如果要操作必须使用ctrl+c，同时redis-server程序结束，不推荐此方法。

```
[root@localhost bin]# ./redis-server
```

下面是启动界面(这个界面只能启动，启动后不能进行其他操作)

```
39451:M 29 May 2020 10:25:41.309 * Increased maximum number of open files to 10032 (it was originally set to 1024).

Redis 5.0.5 (00000000/0) 64 bit
Running in standalone mode
Port: 6379
PID: 39451

http://redis.io

39451:M 29 May 2020 10:25:41.335 # WARNING: The TCP backlog setting of 511 cannot be enforced because /proc/sys/net/core/somaxconn is set to the lower value of 128.
39451:M 29 May 2020 10:25:41.336 # Server initialized
39451:M 29 May 2020 10:25:41.336 # 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.
39451:M 29 May 2020 10:25:41.336 # 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.
39451:M 29 May 2020 10:25:41.336 * Ready to accept connections
```

使用ctrl+c退出前端启动。

3.3.2 Redis的后端启动

修改redis.conf配置文件，设置:daemonize yes,然后可以使用后端模式启动。

```
[root@localhost redis]# vi redis.conf
```

```
##### GENERAL #####

# By default Redis does not run as a daemon. Use 'yes' to run as a daemon.
# Note that Redis will write a pid file in /var/run/redis.pid when running as a daemon.
daemonize yes
```

启动时，指定配置文件(这里所在文件夹是redis)


```
[root@localhost redis]# ./bin/redis-server ./redis.conf
```

Redis默认端口:6379,通过当前服务进行查看

```
[root@localhost redis]# ps -ef | grep -i redis
```

```
[root@localhost redis]# ps -ef | grep -i redis
root      39466      1    0 10:30 ?        00:00:00 ./bin/redis-server 127.0.0.1:6379
root      39471  12847    0 10:31 pts/1    00:00:00 grep -i redis
```

3.3.3 客户端访问redis

如果想要通过指令来操作redis, 可以使用redis的客户端进行操作,在bin文件夹下运行redis-cli

该指令默认连接的127.0.0.1 , 端口号是6379

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

如果想要连接指定的ip地址以及端口号, 则需要按照

```
redis-cli -h ip地址 -p 端口号
```

语法结构连接。

3.3.4 向Redis服务器发送命令

Ping, 测试客户端与Redis的连接是否正常, 如果连接正常, 回收到pong

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

3.3.5 退出客户端

```
127.0.0.1:6379> quit
```

3.3.6 Redis的停止

(1) 强制结束程序。强制终止Redis进程可能会导致redis持久化数据丢失。

语法:kill -9 pid

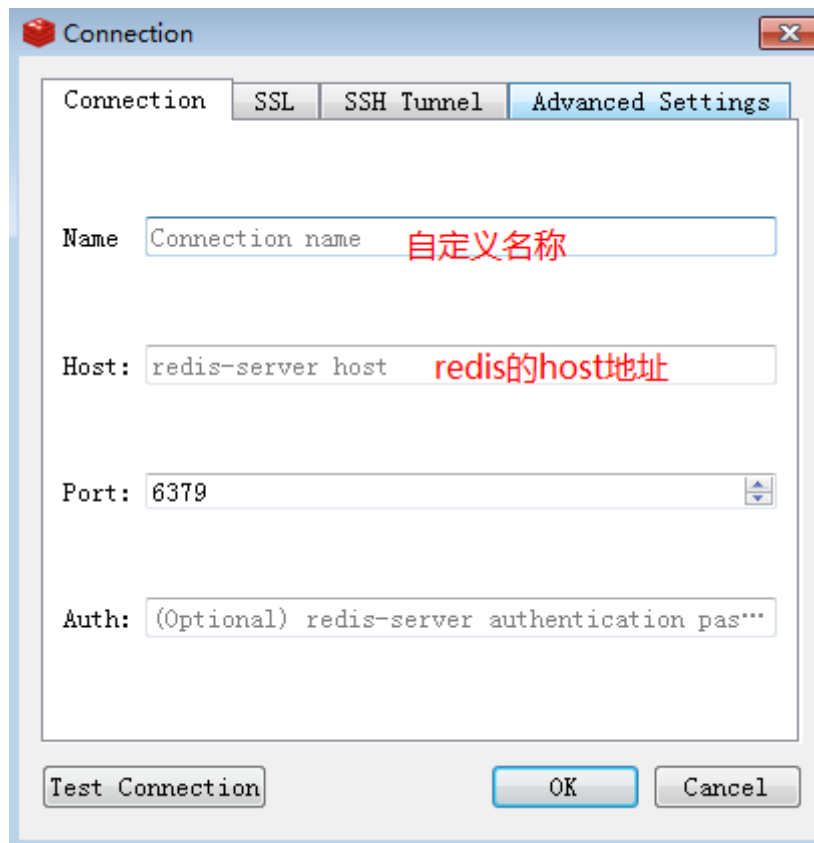
```
kill -9 31475      #pid需要通过“ps aux | grep -i redis”进行查询
```

(2) 正确停止Redis的方式应该是向Redis发送SHUTDOWN命令, 方法为 (关闭默认的端口)

```
[root@localhost redis]# ./bin/redis-cli shutdown
```

```
[root@localhost redis]# ./bin/redis-cli shutdown
[root@localhost redis]# ps -ef | grep -i redis
root      39474  12847    0 10:34 pts/1    00:00:00 grep -i redis
```

3.3.7 第三方工具(redis-desktop-manager)操作redis



注意:需要关闭linux防火墙并且修改redis.conf文件中的bind参数

```
bind linux的ip地址
```

此时如果通过redis客户端访问的时候，代码如下：

```
./redis-cli -h 192.168.197.132 -p 6379
```

4.Redis数据结构

Redis是一种基于内存的数据库，并且提供一定的持久化功能，它是一种键值（key-value）数据库，使用 key 作为索引找到当前缓存的数据，并且返回给程序调用者。

当前的 Redis 支持 6 种数据类型，它们分别是字符串（String）、列表（List）、集合（set）、哈希结构（hash）、有序集合（zset）和基数（HyperLogLog）

5.Redis常用指令

命令学习网站：<http://doc.redisfans.com/index.html>

5.1 String 类型

赋值语法：SET key value

```
127.0.0.1:6379> set k1 zhangsan  
OK
```

取值语法：GET key


```
127.0.0.1:6379> get k1
"zhangsan"
```

设置多个键语法: MSET key value [key value ...]

```
127.0.0.1:6379> mset k2 lisi k3 wangwu
OK
```

获取多个键值语法: MGET key [key ...]

```
127.0.0.1:6379> mget k2 k3
1) "lisi"
2) "wangwu"
```

删除语法: DEL key

```
127.0.0.1:6379> del k3
(integer) 1
127.0.0.1:6379> get k3
(nil)
```

5.2 字符串数字的递增与递减

递增数字: 当存储的字符串是整数时, Redis提供了一个实用的命令INCR, 其作用是让当前键值递增, 并返回递增后的值。

递增数字语法: INCR key

递减数值语法: DECR key

增加指定的整数语法: INCRBY key increment

减少指定的整数 语法: DECRBY key decrement

递增

```
127.0.0.1:6379> incr num
(integer) 1
127.0.0.1:6379> incr num
(integer) 2
```

递减

```
127.0.0.1:6379> decr num
(integer) 1
```

指定步长: (了解)

```
127.0.0.1:6379> incrby num2 2
(integer) 2
127.0.0.1:6379> incrby num2 3
(integer) 5
127.0.0.1:6379> decrby num2 2
(integer) 3
127.0.0.1:6379> decrby num2 1
(integer) 2
```

5.3 Hash散列（了解）

hash叫散列类型，它提供了字段和字段值的映射。字段值只能是字符串类型，不支持散列类型、集合类型等其它类型。相当于是对象格式的存储

赋值语法：HSET key field value

- 设置一个字段值，HSET命令不区分插入和更新操作，当执行插入操作时HSET命令返回1，当执行更新操作时返回0.

```
127.0.0.1:6379> hset user1 username zhangsan
(integer) 1
127.0.0.1:6379> hset user1 username lisi
(integer) 0
```

取值语法：HGET key field

```
127.0.0.1:6379> hget user1 username
"lisi"
```

设置多个字段语法：HMSET key field value [field value ...]

```
127.0.0.1:6379> hmset user1 password 123 age 20
OK
```

取多个值语法: HMGET key field [field ...]

```
127.0.0.1:6379> hmget user1 password age
1) "123"
2) "20"
```

获取所有字段值语法：HGETALL key

```
127.0.0.1:6379> hgetall user1
1) "username"
2) "lisi"
3) "password"
4) "123"
5) "age"
6) "20"
```

删除字段语法：HDEL key field [field ...]

```
127.0.0.1:6379> hdel user1 username
(integer) 1
127.0.0.1:6379> hgetall user1
1) "password"
2) "123"
3) "age"
4) "20"
```

可以支持部分修改。

5.4 队列List

Redis的list是采用来链表来存储,双向链表存储数据, 特点: 增删快、查询慢(Linkedlist).这个队列是有序的。

向列表左边增加元素: LPUSH key value [value ...]

从列表左边弹出元素: LPOP key(临时存储, 弹出后,从队列中清除)

```
127.0.0.1:6379> lpush alist a1 a2 123
(integer) 3
127.0.0.1:6379> lpop alist
"123"
127.0.0.1:6379> lpop alist
"a2"
```

向列表右边增加元素: RPUSH key value [value ...]

从列表右边弹出元素: RPOP key

```
127.0.0.1:6379> rpush blist a1 a2 345
(integer) 3
127.0.0.1:6379> rpop blist
"345"
```

获取列表中元素的个数: LLEN key

```
127.0.0.1:6379> llen blist
(integer) 1
```

查看列表语法: LRANGE key start stop

```
127.0.0.1:6379> lrange blist 0 3
1) "a2"
```

- 将返回start、stop之间的所有元素(包含两端的元素),索引从0开始,可以是负数,如:“-1”代表最后的一个元素。

示例: LPUSH comment1 '{"id":1,"name":"商品","date":1430295077289}'

临时存储。先进先出。使用双向链表:

1, 左边进, 右边去

```
127.0.0.1:6379> lpush stulist stu1
(integer) 1
```

```
127.0.0.1:6379> lpush stulist stu2
(integer) 2
127.0.0.1:6379> lpush stulist stu3
(integer) 3
127.0.0.1:6379> lpush stulist stu4
(integer) 4
127.0.0.1:6379> lpush stulist stu4
(integer) 5
127.0.0.1:6379> lpush stulist stu5
(integer) 6
127.0.0.1:6379> rpop stulist
"stu1"
127.0.0.1:6379> rpop stulist
"stu2"
127.0.0.1:6379> rpop stulist
"stu3"
127.0.0.1:6379> rpop stulist
"stu4"
```

2, 右边进, 左边去。

```
127.0.0.1:6379> rpush clist stu1
(integer) 1
127.0.0.1:6379> rpush clist stu2
(integer) 2
127.0.0.1:6379> rpush clist stu3
(integer) 3
127.0.0.1:6379> rpush clist stu4
(integer) 4
127.0.0.1:6379> rpush clist stu5
(integer) 5
127.0.0.1:6379> lpop clist
"stu1"
127.0.0.1:6379> lpop clist
"stu2"
```

5.5 Set集合

Set集合类型：无序、不可重复

增加元素语法：SADD key member [member ...]

删除元素语法：SREM key member [member ...]

获得集合中的所有元素：SMEMBERS key

```
127.0.0.1:6379> sadd ulist user1
(integer) 1
127.0.0.1:6379> sadd ulist user2
(integer) 1
127.0.0.1:6379> sadd ulist user3
(integer) 1
127.0.0.1:6379> smembers ulist
1) "user2"
2) "user3"
3) "user1"
127.0.0.1:6379> srem ulist user2
(integer) 1
```

```
127.0.0.1:6379> smembers u1ist
1) "user3"
2) "user1"
```

判断元素是否在集合中: SISEMEMBER key member

```
127.0.0.1:6379> smembers u1ist
1) "user3"
2) "user1"
127.0.0.1:6379> sismember u1ist user2
(integer) 0
127.0.0.1:6379> sismember u1ist user1
(integer) 1
```

5.6 Zset有序集合(了解)

Sortedset又叫zset,是有序集合,可排序的,但是唯一。Sortedset和set的不同之处,是会给set中的元素添加一个分数,然后通过这个分数进行排序。

增加元素: ZADD key score member [score member ...]

- 向有序集合中加入一个元素和该元素的分数(score), 如果该元素已经存在则会用新的分数替换原有的分数。

```
127.0.0.1:6379> zadd num1 20 stu1 30 stu2 40 stu3
(integer) 3
```

添加带分数(可用学生成绩, 销售数量等来做分数,方便计算排序):

获得排名在某个范围的元素列表,并按照元素分数降序返回

语法: ZREVRANGE key start stop [WITHSCORES]

```
127.0.0.1:6379> zadd num1 10 stu4
(integer) 1
127.0.0.1:6379> zrevrange num1 0 4
1) "stu3"
2) "stu2"
3) "stu1"
4) "stu4"
```

获取元素的分数 :ZSCORE key member

```
127.0.0.1:6379> zscore num1 stu2
"30"
```

删除元素ZREM key member [member ...]

```
127.0.0.1:6379> zrem num1 stu2
(integer) 1

127.0.0.1:6379> zrevrange num1 0 4
1) "stu3"
2) "stu1"
3) "stu4"
```

获得元素的分数的可以在命令尾部加上WITHSCORES参数

```
127.0.0.1:6379> zrevrange num1 0 4 withscores
1) "stu3"
2) "40"
3) "stu1"
4) "20"
5) "stu4"
6) "10"
```

应用：商品销售量；学生排名等

给某一个属性加分或减分，减分时使用负数：

```
127.0.0.1:6379> zincrby num1 2 stu1
"22"
```

示例：

商品编号1001的销量是9，商品编号1002的销量是10

```
ZADD sellsort 9 1001 10 1002
```

商品编号1001的销量加1

```
ZINCRBY sellsort 1 1001
```

商品销量排序队列中前3名：

```
zrevrange sellsort 0 2 withscores
```

学生排名前3名：

```
zrevrange stus 0 2 withscores
```

5.7 HyperLogLog命令

HyperLogLog是一种使用随机化的算法，以少量内存提供集合中唯一元素数量的近似值。

HyperLogLog 可以接受多个元素作为输入，并给出输入元素的基数估算值：

- **基数**：集合中不同元素的数量。比如 {'apple', 'banana', 'cherry', 'banana', 'apple'} 的基数就是 3。
- **估算值**：算法给出的基数并不是精确的，可能会比实际稍微多一些或者稍微少一些，但会控制在合理的范围之内。

HyperLogLog 的优点是，即使输入元素的数量或者体积非常非常大，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。

HyperLogLog 相关的一些基本命令。

命令	说明
PFADD key element [element ...]	将指定的元素添加到指定的HyperLogLog 中
PFCOUNT key [key ...]	返回给定 HyperLogLog 的基数估算值
PFMERGE destkey sourcekey [sourcekey ...]	将多个 HyperLogLog 合并为一个 HyperLogLog

示例:

```
redis 127.0.0.1:6379> PFADD mykey "redis"
1) (integer) 1
redis 127.0.0.1:6379> PFADD mykey "java"
1) (integer) 1
redis 127.0.0.1:6379> PFADD mykey "mysql"
1) (integer) 1
redis 127.0.0.1:6379> PFCOUNT mykey
(integer) 3
```

5.8 其他命令

(1) keys返回满足给定pattern 的所有key

```
keys user*    //查询以user开头的key
keys *        //查询所有的key
```

(2) exists确认一个key 是否存在,存在返回1

```
127.0.0.1:6379> exists num2    //语法:exists key
(integer) 1
127.0.0.1:6379> exists num23
(integer) 0
```

(3) del删除一个key

```
127.0.0.1:6379> del num1    //语法: del key 删除存在的key返回1, 不存在的key返回0
(integer) 1
127.0.0.1:6379> del num23
(integer) 0
```

(4) rename重命名key:rename oldkey newkey

```
127.0.0.1:6379> rename k1 k11
OK
127.0.0.1:6379> keys *
1) "ulist"
2) "k2"
3) "user1"
4) "num2"
5) "clist"
6) "k11"
```

(5) type返回值的类型: type key


```
127.0.0.1:6379> type ulist
set
127.0.0.1:6379> type kll
string
127.0.0.1:6379> type alist
list
```

设置key的生存时间:缓存的数据一般都是需要设置生存时间的,即:到期后数据销毁。

(6) EXPIRE key seconds

- 设置key的生存时间(单位:秒) key在多少秒后会自动删除
- TTL key 查看key剩余的生存时间
- PERSIST key 清除生存时间

```
127.0.0.1:6379> set a1 123
OK
127.0.0.1:6379> get a1
"123"
127.0.0.1:6379> expire a1 60
(integer) 1
127.0.0.1:6379> ttl a1
(integer) 56
127.0.0.1:6379> ttl a1
(integer) 51
127.0.0.1:6379> ttl a1
(integer) 47
```

(7) 获取服务器信息和统计:info

(8) 删除当前选择数据库中的所有key: flushdb

(9) 删除所有数据库中的所有key:flushall

5.9 Redis的多数据库

一个redis实例key包括多个数据库,客户端可以指定连接某个redis实例的哪个数据库,就好比一个mysql中创建多个数据库,客户端连接时指定连接哪个数据库。

一个redis实例最多可提供16个数据库,下标从0-15,客户端默认连接第0号数据库,也可以通过select选择连接哪个数据库,如下连接1号库:

```
127.0.0.1:6379[1]> select 1
OK
127.0.0.1:6379[1]> keys *
(empty list or set)
```

切换至0数据库下面

```
127.0.0.1:6379[1]> select 0
OK
127.0.0.1:6379> keys *
1) "username"
2) "num"
3) "company"
4) "mylist"
```

将key的数据移动到1号数据库: move key 数据库编号

```
127.0.0.1:6379[1]> select 0
OK
127.0.0.1:6379> keys *
1) "username"
2) "num"
3) "company"
4) "mylist"
127.0.0.1:6379> move username 1
(integer) 1
127.0.0.1:6379> keys *
1) "num"
2) "company"
3) "mylist"
127.0.0.1:6379> select 1
OK
127.0.0.1:6379[1]> keys username
1) "username"
```

6.Redis的事务管理

Redis 事务可以一次执行多个命令，并且带有以下两个重要的保证：

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

事务是一个原子操作：事务中的命令要么全部被执行，要么全部都不执行。

一个事务从开始到执行会经历以下三个阶段：

开始事务。

命令入队。

执行事务。

实例

以下是一个事务的例子，它先以 MULTI 开始一个事务，然后将多个命令入队到事务中，最后由 EXEC 命令触发事务，一并执行事务中的所有命令：

示例1:

```
127.0.0.1:6379> multi
OK
127.0.0.1:6379> set bookname java
QUEUED
127.0.0.1:6379> set bookname c++
QUEUED
127.0.0.1:6379> set bookname html
QUEUED
127.0.0.1:6379> exec
1) OK
2) OK
3) OK
```

示例2:

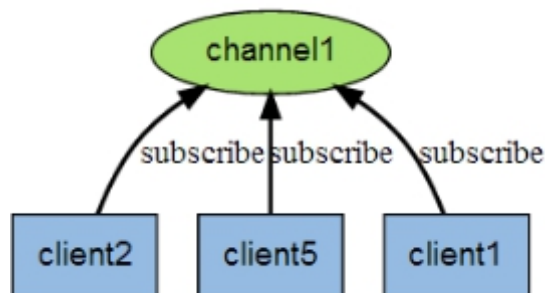
```
127.0.0.1:6379> multi
OK
127.0.0.1:6379> set u1 user1
QUEUED
127.0.0.1:6379> get u1
QUEUED
127.0.0.1:6379> sadd tag c++ html java
QUEUED
127.0.0.1:6379> smembers tag
QUEUED
127.0.0.1:6379> exec
1) OK
2) "user1"
3) (integer) 3
4) 1) "java"
   2) "html"
   3) "c++"
```

7.Redis发布订阅模式

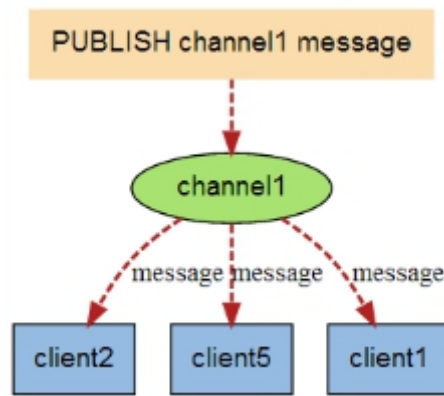
Redis 发布订阅(pub/sub)是一种消息通信模式：发送者(pub)发送消息，订阅者(sub)接收消息。

Redis 客户端可以订阅任意数量的频道。

下图展示了频道 channel1，以及订阅这个频道的三个客户端 —— client2、client5 和 client1 之间的关系：



当有新消息通过 PUBLISH 命令发送给频道 channel1 时，这个消息就会被发送给订阅它的三个客户端：



在我们实例中我们创建了订阅频道名为 redisMessage:

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

现在, 我们先重新开启个 redis 客户端, 然后在同一个频道 redisMessage 发布两次消息, 订阅者就能接收到消息。

```
127.0.0.1:6379> publish redisMessage "demo1 test"
(integer) 1
127.0.0.1:6379> publish redisMessage "demo2 test"
(integer) 1
127.0.0.1:6379> publish redisMessage "demo3 test"
(integer) 1
```

订阅者的客户端会显示如下消息

```
127.0.0.1:6379> subscribe redisMessage
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "redisMessage"
3) (integer) 1
1) "message"
2) "redisMessage"
3) "demo1 test"
1) "message"
2) "redisMessage"
3) "demo2 test"
1) "message"
2) "redisMessage"
3) "demo3 test"
```

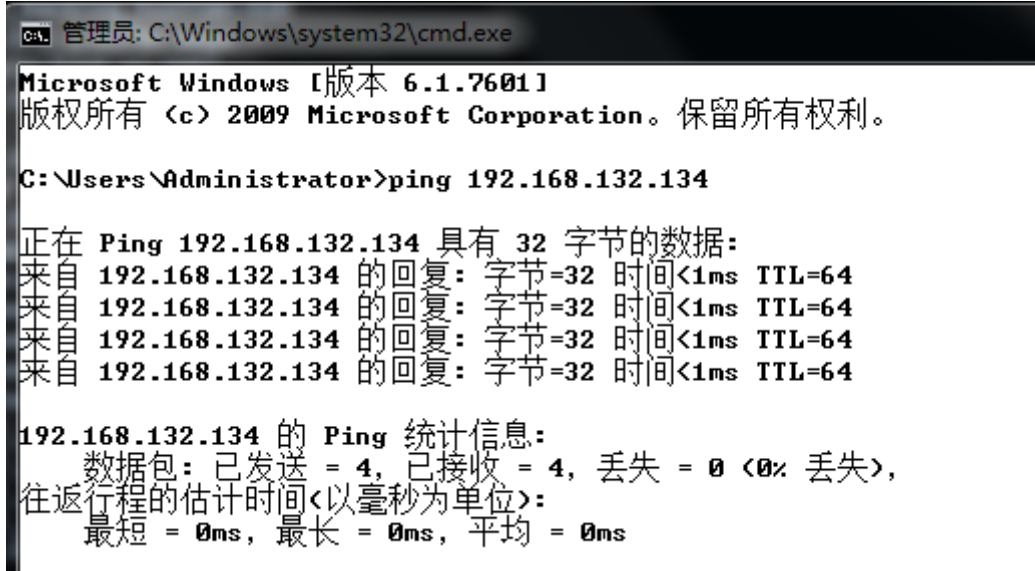
8.Jedis连接Redis

第一步: 创建项目, 导入依赖

```
<dependency>
  <groupId>redis.clients</groupId>
  <artifactId>jedis</artifactId>
  <version>2.7.2</version>
</dependency>
```

注意:

1) 确认远程服务器是否可以ping通: ping vm的ip地址



```
C:\Windows\system32\cmd.exe
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

C:\Users\Administrator>ping 192.168.132.134

正在 Ping 192.168.132.134 具有 32 字节的数据:
来自 192.168.132.134 的回复: 字节=32 时间<1ms TTL=64
来自 192.168.132.134 的回复: 字节=32 时间<1ms TTL=64
来自 192.168.132.134 的回复: 字节=32 时间<1ms TTL=64
来自 192.168.132.134 的回复: 字节=32 时间<1ms TTL=64

192.168.132.134 的 Ping 统计信息:
    数据包: 已发送 = 4, 已接收 = 4, 丢失 = 0 (0% 丢失),
    往返行程的估计时间<以毫秒为单位>:
        最短 = 0ms, 最长 = 0ms, 平均 = 0ms
```

2)确认防火墙是否关闭或放行

service iptables stop

service iptables status

第二步: 链接服务器

方案一

单实例链接

```
Jedis jedis = new Jedis("ip地址", 端口号); //建立链接
```

核心代码:

```
public static void main(String[] args) {
    Jedis jedis=new Jedis("192.168.197.129",6379);
    //设置值
    jedis.set("java001","java工程师");
    String java001 = jedis.get("java001");
    System.out.println(java001);
}
```

常见异常:

```
Exception in thread "main" redis.clients.jedis.exceptions.JedisConnectionException: java.net.ConnectException: Connection refused: connect
    at redis.clients.jedis.Connection.connect(Connection.java:155)
    at redis.clients.jedis.BinaryClient.connect(BinaryClient.java:83)
    at redis.clients.jedis.Connection.sendCommand(Connection.java:94)
    at redis.clients.jedis.BinaryClient.set(BinaryClient.java:100)
    at redis.clients.jedis.Client.set(Client.java:29)
    at redis.clients.jedis.Jedis.set(Jedis.java:65)
    at com.xzk.Demo.main(Demo.java:11)
```

解决方案:

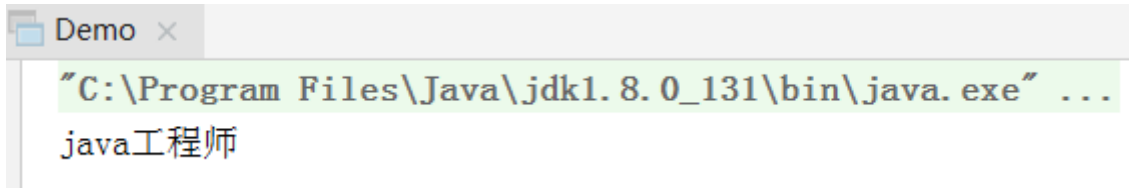
虚拟机客户端连接的ip是127.0.0.1,意思是连接的本机,其他机器无法连接,这里需要修改配置文件,将连接地址改为虚拟机的地址,就可以了.

修改redis.conf文件里面的 bind 连接地址,将连接地址改为自己虚拟机的ip

```
bind 192.168.197.129
```

重新启动服务,jedis就可以正常连上了

Idea中控制台打印:



The screenshot shows an IDE window titled 'Demo' with a console output. The first line is a command: `"C:\Program Files\Java\jdk1.8.0_131\bin\java.exe" ...`. The second line is the output: `java工程师`.

服务器上存储:

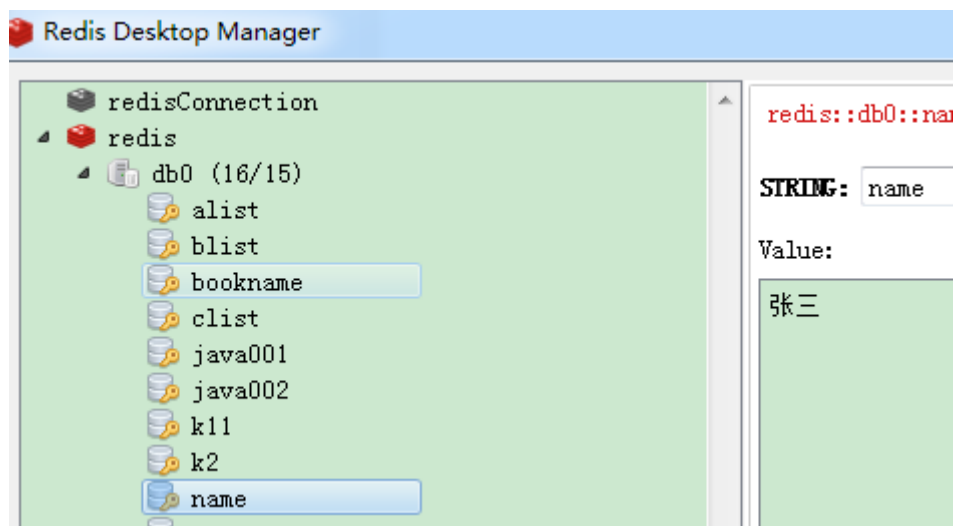


方案二：连接池

jedis连接池连接,后面会使用Spring的配置文件来整合。

```
// 1.获取连接池配置对象,设置配置项
JedisPoolConfig config = new JedisPoolConfig();
// 1.1最大的连接数
config.setMaxTotal(30);
// 1.2最大的空闲
config.setMaxIdle(10);
// 2.获取连接池
JedisPool jedisPool = new JedisPool(config, "192.168.197.129", 6379);
Jedis jedis = null;
try {
    jedis = jedisPool.getResource();
    // 3.设置数据
    jedis.set("name", "张三");
    String name = jedis.get("name");
    System.out.println("name=" + name);
} catch (Exception e) {
    e.printStackTrace();
} finally {
    if (jedis != null) {
        jedis.close();
    }
    // 4.虚拟机关闭的时候,释放资源
    if (jedisPool != null) {
        jedisPool.close();
    }
}
```

服务端存储确认：



9.Redis持久化方式

9.1 什么是Redis持久化

由于redis的值放在内存中，为防止突然断电等特殊情况的发生，需要对数据进行持久化备份。即将内存数据保存到硬盘。

9.2 Redis 持久化存储方式

9.2.1 RDB持久化

RDB 是以二进制文件，是在某个时间点将数据写入一个临时文件，持久化结束后，用这个临时文件替换上次持久化的文件，达到数据恢复。

优点：使用单独子进程来进行持久化，主进程不会进行任何 IO 操作，保证了 redis 的高性能

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

这里说的这个执行数据写入到临时文件的时间点是可以通过配置来自己确定的，通过配置redis 在 n 秒内如果超过 m 个 key 被修改这执行一次 RDB 操作。这个操作就类似于在这个时间点来保存一次 Redis 的所有数据，一次快照数据。所有这个持久化方法也通常叫做 snapshots。

RDB 默认开启，redis.conf 中的具体配置参数如下；

```
#dbfilename: 持久化数据存储在本地的文件
dbfilename dump.rdb
#dir: 持久化数据存储在本地的路径，如果是在/redis/redis-5.0.5/src下启动的redis-cli，则数据
会存储在当前src目录下
dir ./
##snapshot触发的时机，save
##如下为900秒后，至少有一个变更操作，才会snapshot
##对于此值的设置，需要谨慎，评估系统的变更操作密集程度
##可以通过“save”来关闭snapshot功能
#save时间，以下分别表示更改了1个key时间间隔900s进行持久化存储；更改了10个key300s进行存储；更改
10000个key60s进行存储。
save 900 1
save 300 10
save 60 10000
```



```
##当snapshot时出现错误无法继续时，是否阻塞客户端“变更操作”，“错误”可能因为磁盘已满/磁盘故障/OS级别异常等
stop-writes-on-bgsave-error yes
##是否启用rdb文件压缩，默认为“yes”，压缩往往意味着“额外的cpu消耗”，同时也意味这较小的文件尺寸以及较短的网络传输时间
rdbcompression yes
```

注意:测试时使用root用户操作

9.2.2 AOF持久化

Append-Only File，将“操作 + 数据”以格式化指令的方式追加到操作日志文件的尾部，在 append 操作返回后(已经写入到文件或者将要写入)，才进行实际的数据变更，“日志文件”保存了历史所有的操作过程；当 server 需要数据恢复时，可以直接 replay 此日志文件，即可还原所有的操作过程。AOF 相对可靠，AOF 文件内容是字符串，非常容易阅读和解析。

优点：可以保持更高的数据完整性，如果设置追加 file 的时间是 1s，如果 redis 发生故障，最多会丢失 1s 的数据；且如果日志写入不完整支持 redis-check-aof 来进行日志修复；AOF 文件没被 rewrite 之前（文件过大时会对命令进行合并重写），可以删除其中的某些命令（比如误操作的 flushall）。

缺点：AOF 文件比 RDB 文件大，且恢复速度慢。

我们可以简单的认为 AOF 就是日志文件，此文件只会记录“变更操作”(例如：set/del 等)，如果 server 中持续的大量变更操作，将会导致 AOF 文件非常的庞大，意味着 server 失效后，数据恢复的过程将会很长；事实上，一条数据经过多次变更，将会产生多条 AOF 记录，其实只要保存当前的状态，历史的操作记录是可以抛弃的；因为 AOF 持久化模式还伴生了“AOF rewrite”。

AOF 的特性决定了它相对比较安全，如果你期望数据更少的丢失，那么可以采用 AOF 模式。如果 AOF 文件正在被写入时突然 server 失效，有可能导致文件的最后一次记录是不完整，你可以通过手工或者程序的方式去检测并修正不完整的记录，以便通过 aof 文件恢复能够正常；同时需要提醒，如果你的 redis 持久化手段中有 aof，那么在 server 故障失效后再次启动前，需要检测 aof 文件的完整性。

AOF 默认关闭，开启方法，修改配置文件 redis.conf：appendonly yes

```
##此选项为aof功能的开关，默认为“no”，可以通过“yes”来开启aof功能
##只有在“yes”下，aof重写/文件同步等特性才会生效
appendonly yes

##指定aof文件名称
appendfilename appendonly.aof

##指定aof操作中文件同步策略，有三个合法值：always everysec no,默认为everysec
appendfsync everysec
##在aof-rewrite期间，appendfsync是否暂缓文件同步，“no”表示“不暂缓”，“yes”表示“暂缓”，默认为“no”
no-appendfsync-on-rewrite no

##aof文件rewrite触发的最小文件尺寸(mb,gb)，只有大于此aof文件大于此尺寸是才会触发rewrite，默认“64mb”，建议“512mb”
auto-aof-rewrite-min-size 64mb

##相对于“上一次”rewrite，本次rewrite触发时aof文件应该增长的百分比。
##每一次rewrite之后，redis都会记录下此时“新aof”文件的大小(例如A)，那么当aof文件增长到A*(1 + p)之后
##触发下一次rewrite，每一次aof记录的添加，都会检测当前aof文件的尺寸。
auto-aof-rewrite-percentage 100
```

AOF 是文件操作，对于变更操作比较密集的 server，那么必将造成磁盘 IO 的负荷加重；此外 linux 对文件操作采取了“延迟写入”手段，即并非每次 write 操作都会触发实际磁盘操作，而是进入了 buffer 中，当 buffer 数据达到阈值时触发实际写入(也有其他时机)，这是 linux 对文件系统的优化，但是这却有可能带来隐患，如果 buffer 没有刷新到磁盘，此时物理机器失效(比如断电)，那么有可能导致最后一条或者多条 aof 记录的丢失。通过上述配置文件，可以得知 redis 提供了 3 中 aof 记录同步选项：

always：每一条 aof 记录都立即同步到文件，这是最安全的方式，也以为更多的磁盘操作和阻塞延迟，是 IO 开支较大。

everysec：每秒同步一次，性能和安全都比较中庸的方式，也是 redis 推荐的方式。如果遇到物理服务器故障，有可能导致最近一秒内 aof 记录丢失(可能为部分丢失)。

no：redis 并不直接调用文件同步，而是交给操作系统来处理，操作系统可以根据 buffer 填充情况 / 通道空闲时间等择机触发同步；这是一种普通的文件操作方式。性能较好，在物理服务器故障时，数据丢失量会因 OS 配置有关。

其实，我们可以选择的太少，everysec 是最佳的选择。如果你非常在意每个数据都极其可靠，建议你选择一款“关系性数据库”。

9.2.3 AOF与RDB区别

RDB:

RDB是在某个时间点将数据写入一个临时文件，持久化结束后，用这个临时文件替换上次持久化的文件，达到数据恢复。

优点：使用单独子进程来进行持久化，主进程不会进行任何IO操作，保证了redis的高性能

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

AOF:

Append-only file，将“操作 + 数据”以格式化指令的方式追加到操作日志文件的尾部，在append操作返回后(已经写入到文件或者即将写入)，才进行实际的数据变更，“日志文件”保存了历史所有的操作过程；当server需要数据恢复时，可以直接replay此日志文件，即可还原所有的操作过程。AOF相对可靠，它和mysql中bin.log、apache.log、zookeeper中txn-log简直异曲同工。AOF文件内容是字符串，非常容易阅读和解析。

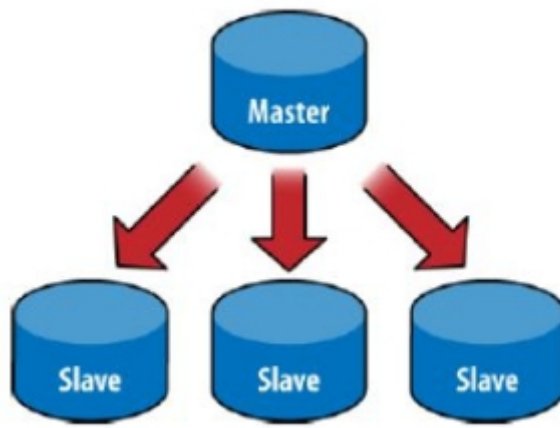
优点：可以保持更高的数据完整性，如果设置追加file的时间是1s，如果redis发生故障，最多会丢失1s的数据；且如果日志写入不完整支持redis-check-aof来进行日志修复；AOF文件没被rewrite之前（文件过大时会命令进行合并重写），可以删除其中的某些命令（比如误操作的flushall）。

缺点：AOF文件比RDB文件大，且恢复速度慢。

10.Redis主从复制

- 持久化保证了即使redis服务重启也不会丢失数据，但是当redis服务器的硬盘损坏了可能会导致数据丢失，通过redis的主从复制机制就可以避免这种单点故障（单台服务器的故障）。
- 主redis中的数据和从上的数据保持实时同步,当主redis写入数据时通过主从复制机制复制到两个从服务上。
- 主从复制不会阻塞master，在同步数据时，master可以继续处理client请求。
- 主机master配置:无需配置

推荐主从模式同步数据:



工作中一般选用：一主两从或一主一从

数据会同步到从服务器。在这个集群中的几台服务器上都有同样的数据。

主从搭建步骤:

主机：不用配置。仅仅只需要配置从机,从机slave配置:(这里是伪集群)

第一步：复制出一个从机,注意使用root用户

```
[root@localhost myapps]# cp redis/ redis1 -r
[root@localhost myapps]# ll
总用量 40
drwxr-xr-x. 3 root root 4096 2月 1 09:26 redis
drwxr-xr-x. 3 root root 4096 2月 1 09:27 redis1
```

第二步：修改从机的redis.conf

语法：replicaof // replicaof 主机ip 主机端口号

提示:检索文件: 输入:/replicaof 当前页没有时, 输入n, 查找下一页

第三步：修改从机的port地址为6380

在从机redis.conf中修改

```
##### GENERAL #####

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

# When running daemonized, Redis writes a pid file in /var/run/redis.pid by
# default. You can specify a custom pid file location here.
pidfile /var/run/redis.pid

# Accept connections on the specified port, default is 6379.
# If port 0 is specified Redis will not listen on a TCP socket.
port 6380
```

第四步：清除从机中的持久化文件

```
[root@localhost bin]# rm -rf appendonly.aof dump.rdb
[root@localhost bin]# ll
总用量 15440
-rwxr-xr-x. 1 root root 4588902 7月 1 09:27 redis-benchmark
-rwxr-xr-x. 1 root root 22225 7月 1 09:27 redis-check-aof
-rwxr-xr-x. 1 root root 45443 7月 1 09:27 redis-check-dump
-rwxr-xr-x. 1 root root 4691809 7月 1 09:27 redis-cli
lrwxrwxrwx. 1 root root 12 7月 1 09:27 redis-sentinel -> redis-server
-rwxr-xr-x. 1 root root 6450337 7月 1 09:27 redis-server
```

第五步：启动从机

```
[root@localhost redis1]# ./bin/redis-server ./redis.conf
```

第六步：启动6380的客户端

```
[root@localhost redis1]# ./bin/redis-cli -p 6380
127.0.0.1:6380> keys *
1) "mylist"
2) "num"
3) "bookCate1"
4) "newbook"
127.0.0.1:6380>
```

停止客户端: ./bin/redis-cli -p 6380 shutdown

注意：

Ø 主机一旦发生增删改操作，那么从机会自动将数据同步到从机中

Ø 从机不能执行写操作,只能读

```
127.0.0.1:6380> get username
"hehe"
127.0.0.1:6380> set username haha
(error) READONLY You can't write against a read only slave.
```

复制的过程原理

- 当从库和主库建立MS(master slaver)关系后，会向主数据库发送SYNC命令；
- 主库接收到SYNC命令后会开始在后台保存快照（RDB持久化过程），并将期间接收到的写命令缓存起来；
- 快照完成后,主Redis会将快照文件和所有缓存的写命令发送给从Redis；
- 从Redis接收到后，会载入快照文件并且执行收到的缓存命令；
- 主Redis每当接收到写命令时就会将命令发送从Redis，保证数据的一致；【内部完成,所以**不支持客户端在从机人为写数据。**】

复制架构中出现宕机情况？

从Redis宕机:重启就好

主Redis宕机:从数据库(从机)中执行SLAVEOF NO ONE命令，断开主从关系并且提升为主库继续服务[把一个从做为主机，这个时候新主机[之前的从机]就具备写入的能力]；主服务器修好后，重新启动后，执行SLAVEOF命令，将其设置为从库[老主机设置为从机]。[手动执行，过程复杂，容易出错。]是否有更好的方案？

11.Redis哨兵模式

哨兵模式：给集群分配一个站岗的。

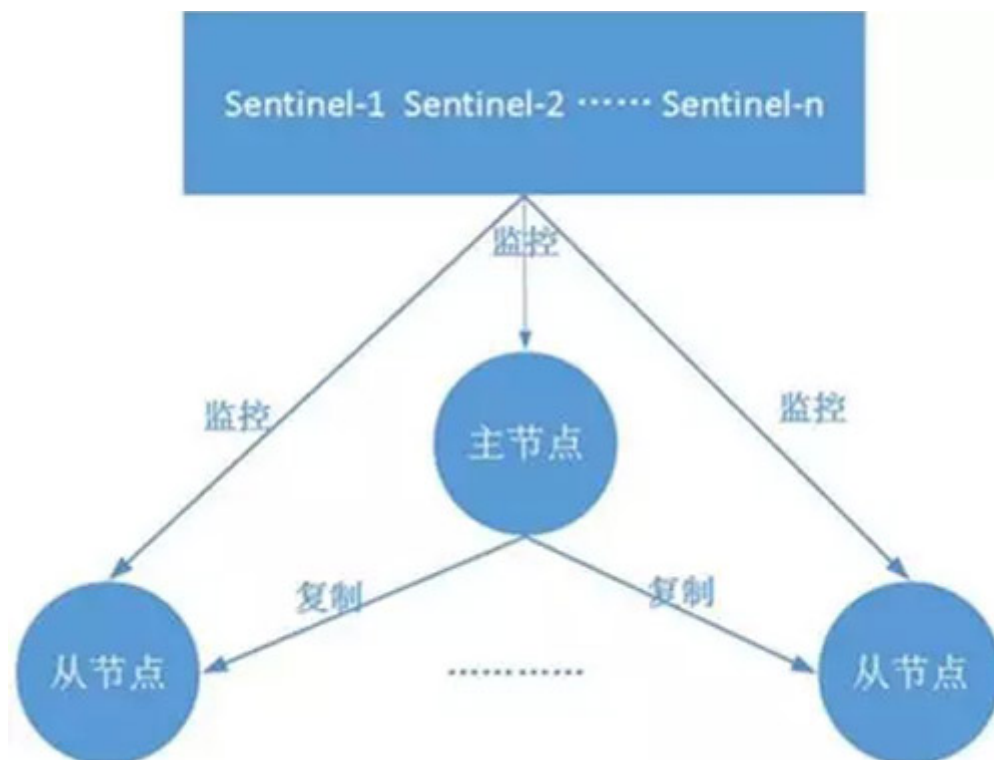
哨兵的作用就是对Redis系统的运行情况监控，它是一个独立进程,它的功能：

1. 监控主数据库和从数据库是否运行正常；
2. 主数据出现故障后自动将从数据库转化为主数据库；

如果主机宕，开启选举工作，选择一个从做主机。

环境准备：一主两从，启动任一从机时，启动哨兵模式

虽然哨兵(sentinel) 释出为一个单独的可执行文件 redis-sentinel ,但实际上它只是一个运行在特殊模式下的 Redis 服务器，你可以在启动一个普通 Redis 服务器时通过给定 --sentinel 选项来启动哨兵(sentinel)。



11.1. 第一步：配置哨兵：

哨兵主要是用来监听主服务器的，所以一般把哨兵部署在从服务器上监听。

配置哨兵

- 启动哨兵进程，首先需要创建哨兵配置文件vi sentinel.conf,可从源码配置redis-5.0.5/sentinel.conf中复制内容，也可以直接自定义该文件到bin目录下
- 在配置中输入:sentinel monitor mastername 内网IP(127.0.0.1) 6379 1
- 说明：
 - mastername 监控主数据的名称，自定义
 - 127.0.0.1： 监控主数据库的IP；
 - 6379:端口
 - 1： 最低通过票数

11.2. 第二步：启动哨兵：

哨兵是一个单独的进程，启动之前确保主从服务是正常的。先启动主服务，后启动从服务

```
redis-sentinel -> redis-server
redis-server
sentinel.conf
```

把日志写入指定的文件

```
[root@localhost bin]# ./redis-sentinel ./sentinel.conf >sent.log &
[1] 3373
```

启动redis服务后，程序会自动配置文件sentinel.conf，并生成内容，注意:若再起启动需要删除下生成的内容。

哨兵启动方式:

```
[root@localhost bin]# ./redis-server redis.conf --sentinel
```

哨兵进程控制台: 为master数据库添加了一个监控.

```
sentinel ID is 0228209c4a6107c32767057312a5a324dc3f7718
+monitor master mymaster 127.0.0.1 6379 quorum 1
+slave slave 172.17.140.3:6378 172.17.140.3 6378 @ mymaster 127.0.0.1 6379
```

同时多了哨兵进程:

```
127.0.0.1:6380> quit
[root@iZm5edxdji7ys1crbmd9gfZ bin]# ps -aux|grep redis
root      19173  0.0  0.2 140892 2192 ?        Ssl  10:27   0:03 ./redis-server 0.0.0.0:6379
root      19177  0.0  0.2 140892 2184 ?        Ssl  10:27   0:04 ./redis-server 0.0.0.0:6380
root      19182  0.0  0.2 140892 2204 ?        Ssl  10:28   0:03 ./redis-server 0.0.0.0:6381
root      19237  0.0  0.1 20204 1240 pts/2    S+   10:34   0:00 ./redis-cli -h 127.0.0.1 -p 6381
root      19335  0.4  0.2 141024 2636 pts/0    Sl+  11:58   0:00 ./redis-server *:26379 [sentinel]
root      19339  0.0  0.0 112708  984 pts/1    R+   11:59   0:00 grep --color=auto redis
```

查询配置文件sentinel.conf中生成的内容:

启动哨兵的时候，修改了哨兵的配置文件。如果需要再次启动哨兵，需要删除myid唯一标示。

(保险的做法就是启动的一次，新配置一次)

```
sentinel myid f88f8325c0f3c3facf07156f96eaf8bc2b0b0180
# Generated by CONFIG REWRITE
port 26379
dir "/home/admin/myapps/redisSentinel/bin"
protected-mode no
sentinel deny-scripts-reconfig yes
sentinel monitor master 192.168.197.129 6379 1
sentinel config-epoch master 2
sentinel leader-epoch master 2
sentinel known-replica master 127.0.0.1 6380
```

11.3. 第三步: 主机宕机

机房意外: 断电了。硬件故障: 硬盘坏了。

列表:

```
[root@localhost redis6380]# ps -aux |grep redis
Warning: bad syntax, perhaps a bogus '-'? See /usr/share/doc/procps-3.2.8/FAQ
root      2910  0.1  1.2 167776 12148 ?        Ssl  18:41   0:07 ./bin/redis-server 127.0.0.1:6380
root      2916  0.1  1.2 161632 12312 ?        Ssl  18:41   0:07 ./bin/redis-server 127.0.0.1:6381
root      2990  0.3  0.8 152416  8244 pts/4    Sl   18:53   0:10 ./redis-sentinel *:26379 [sentinel]
root      3019  0.0  0.7  23864  7324 pts/3    S+   18:57   0:00 ./bin/redis-cli -h 192.168.197.129 -p 6379
root      3029  0.0  0.7  23996  7504 pts/1    S+   18:59   0:00 ./bin/redis-cli -h 127.0.0.1 -p 6380
root      3030  0.0  0.7  23996  7504 pts/2    S+   19:00   0:00 ./bin/redis-cli -h 127.0.0.1 -p 6381
root      3342  0.1  1.6 167776 16248 ?        Ssl  19:25   0:01 ./bin/redis-server 192.168.197.129:6379
```

杀死主机: kill -9 pid

```
[root@localhost redis6380]# kill -9 2910
```

哨兵控制台：从库自动提升为主库。

哨兵工作,链接之前的从机确认:

```
127.0.0.1:6381> info replication
# Replication
role:slave
master_host:192.168.197.129
master_port:6379
```

哨兵替代运维。自动监控完成。

同时也会自动修改redis.conf的主从配置文件。

```
replicaof 127.0.0.1 6380
```

指向了新主机。再次启动原有的主机,原有的主机会变为从机。

总结:

主从集群: 主机有写入权限。从机没有, 只有可读。

意外宕机方案:

手动恢复: 人为重启服务器, 主机宕, 把从机设置为主机。

自动恢复: 使用哨兵监控。自动切换主从。

12.Redis集群方案

12.1 redis-cluster架构图

架构细节:

(1)所有的redis节点彼此互联(PING-PONG机制),内部使用二进制协议优化传输速度和带宽。

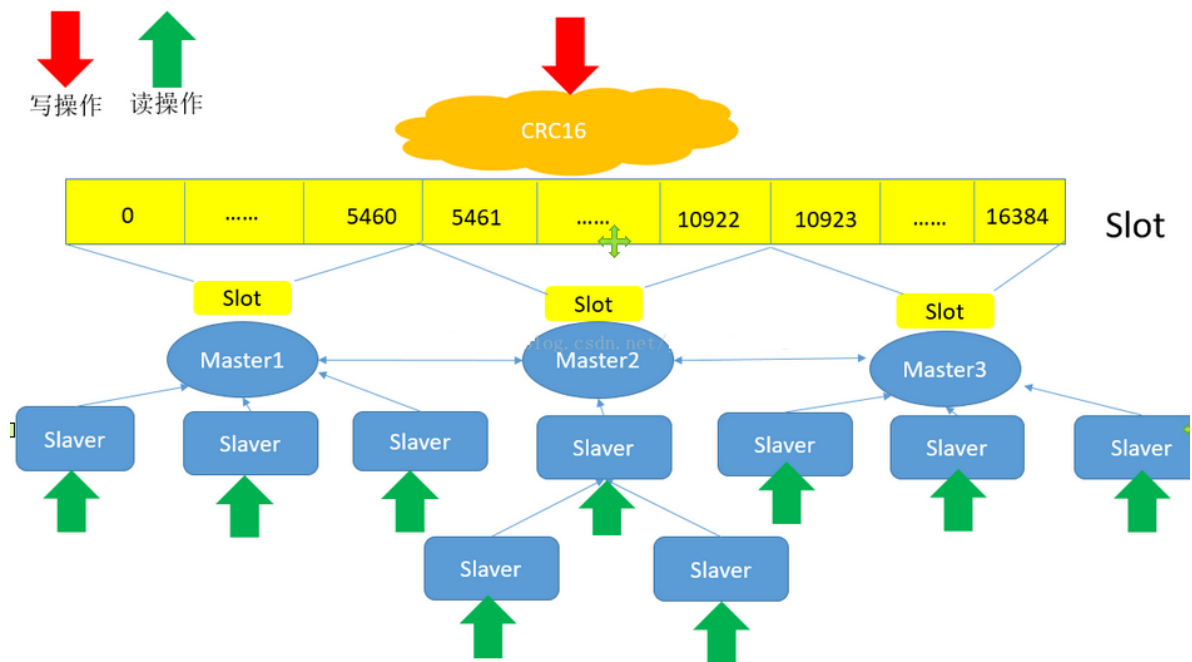
(2)节点的fail是通过集群中超过半数的节点检测有效时整个集群才生效。

(3)客户端与redis节点直连,不需要中间proxy层.客户端不需要连接集群所有节点,连接集群中任何一个可用节点即可

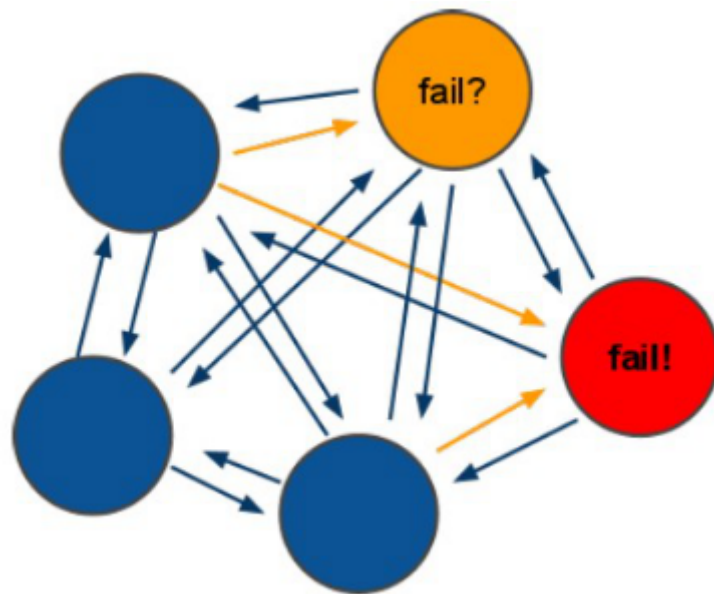
(4)redis-cluster把所有的物理节点映射到[0-16383]slot上,cluster 负责维护node<->slot<->value

Redis 集群中内置了 16384 个哈希槽, 当需要在 Redis 集群中放置一个 key-value 时, redis 先对 key 使用 crc16 算法算出一个结果, 然后把结果对 16384 求余数, 这样每个 key 都会对应一个编号在 0-16383 之间的哈希槽, redis 会根据节点数量大致均等的将哈希槽映射到不同的节点

示例如下:



12.2 redis-cluster投票:容错



心跳机制

(1) 集群中所有master参与投票,如果半数以上master节点与其中一个master节点通信超过(cluster-node-timeout),认为该master节点挂掉.

(2): 什么时候整个集群不可用(cluster_state:fail)?

Ø 如果集群任意master挂掉,且当前master没有slave,则集群进入fail状态。也可以理解成集群的[0-16383]slot映射不完全时进入fail状态。

Ø 如果集群超过半数以上master挂掉,无论是否有slave,集群进入fail状态。

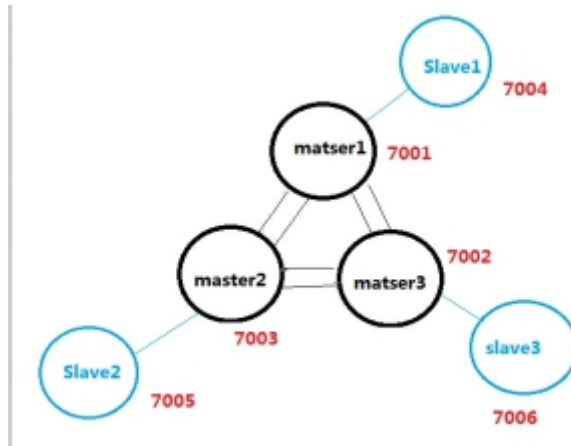
12.3 集群搭建步骤

第一步:安装redis

第二步:创建集群目录

```
[root@localhost redis]# mkdir redis-cluster
```

第三步:在集群目录下创建节点目录



搭建集群最少也得需要3台主机, 如果每台主机再配置一台从机的话, 则最少需要6台机器。设计端口如下: 创建6个redis实例, 需要端口号7001~7006

```
[root@localhost myapps]# cp redis/ redis-cluster/7001 -r
[root@localhost myapps]# cd redis-cluster/7001
[root@localhost 7001]# ll
drwxr-xr-x. 2 root root 4096 7月 1 10:22 bin
-rw-r--r--. 1 root root 3446 7月 1 10:22 dump.rdb
-rw-r--r--. 1 root root 41404 7月 1 10:22 redis.conf
```

第四步: 如果存在持久化文件, 则删除

```
[root@localhost 7001]# rm -rf appendonly.aof dump.rdb
```

第五步: 修改redis.conf配置文件, 打开Cluster-enable yes

说明: cluster-enable 是否支持集群

```
##### REDIS CLUSTER #####
#
# *****
# WARNING EXPERIMENTAL: Redis Cluster is considered to be stable code, however
# in order to mark it as "mature" we need to wait for a non trivial percentage
# of users to deploy it in production.
# *****
#
# Normal Redis instances can't be part of a Redis Cluster; only nodes that are
# started as cluster nodes can. In order to start a Redis instance as a
# cluster node enable the cluster support uncommenting the following:
#
cluster-enabled yes
```

第六步: 修改端口

```
# When running daemonized, Redis writes a pid file in /var/run/redis.pid by
# default. You can specify a custom pid file location here.
pidfile /var/run/redis.pid

# Accept connections on the specified port, default is 6379.
# If port 0 is specified Redis will not listen on a TCP socket.
port 7001
```

第七步：复制出7002-7006机器

```
[root@localhost redis-cluster]# cp 7001/ 7002 -r
[root@localhost redis-cluster]# cp 7001/ 7003 -r
[root@localhost redis-cluster]# cp 7001/ 7004 -r
[root@localhost redis-cluster]# cp 7001/ 7005 -r
[root@localhost redis-cluster]# cp 7001/ 7006 -r
[root@localhost redis-cluster]# ll
total 28
drwxr-xr-x. 3 root root 4096 Jun  2 00:02 7001
drwxr-xr-x. 3 root root 4096 Jun  2 00:02 7002
drwxr-xr-x. 3 root root 4096 Jun  2 00:02 7003
drwxr-xr-x. 3 root root 4096 Jun  2 00:03 7004
drwxr-xr-x. 3 root root 4096 Jun  2 00:03 7005
drwxr-xr-x. 3 root root 4096 Jun  2 00:03 7006
-rwxr-xr-x. 1 root root 3600 Jun  1 23:52 redis-trib.rb
```

第八步：修改7002-7006机器的端口

第九步：启动7001-7006这六台机器，写一个启动脚本：自定义shell脚本

```
[root@localhost redis-cluster]# vi startall.sh
```

内容:

```
cd 7001
./bin/redis-server ./redis.conf
cd ..
cd 7002
./bin/redis-server ./redis.conf
cd ..
cd 7003
./bin/redis-server ./redis.conf
cd ..
cd 7004
./bin/redis-server ./redis.conf
cd ..
cd 7005
./bin/redis-server ./redis.conf
cd ..
cd 7006
./bin/redis-server ./redis.conf
cd ..
```

第十步：修改start-all.sh文件的权限

```
[root@localhost redis-cluster]# chmod u+x startall.sh
```

第十一步：启动所有的实例

```
[root@localhost redis-cluster]# ./startall.sh
```

第十二步：创建集群（关闭防火墙）

注意：在任意一台上运行 不要在每台机器上都运行，一台就够了 redis 5.0.5中使用redis-cli --cluster替代redis-trib.rb,命令如下

```
redis-cli --cluster create ip:port ip:port --cluster-replicas 1
```

```
[root@localhost redis_cluster]# cd /home/admin/myapps/redis-cluster/7001/bin
[root@localhost bin]# ./redis-cli --cluster create 192.168.197.132:7001
192.168.197.132:7002 192.168.197.132:7003 192.168.197.132:7004
192.168.197.132:7005 192.168.197.132:7006 --cluster-replicas 1

\>>> Creating cluster

Connecting to node 127.0.0.1:7001: OK
Connecting to node 127.0.0.1:7002: OK
Connecting to node 127.0.0.1:7003: OK
Connecting to node 127.0.0.1:7004: OK
Connecting to node 127.0.0.1:7005: OK
Connecting to node 127.0.0.1:7006: OK
\>>> Performing hash slots allocation on 6 nodes...
Using 3 masters:
127.0.0.1:7001
127.0.0.1:7002
127.0.0.1:7003

Adding replica 127.0.0.1:7004 to 127.0.0.1:7001
Adding replica 127.0.0.1:7005 to 127.0.0.1:7002
Adding replica 127.0.0.1:7006 to 127.0.0.1:7003

[OK] All 16384 slots covered.
```

12.4 连接集群

命令:

```
[root@localhost 7001]# ./bin/redis-cli -h 127.0.0.1 -p 7001 -c
```

-c: 指定是集群连接

```
[root@localhost 7001]# ./bin/redis-cli -h 127.0.0.1 -p 7001 -c
127.0.0.1:7001> set username java123
-> Redirected to slot [14315] located at 127.0.0.1:7003
OK
```

关闭防火墙:service iptables stop

查看防火墙状态:service iptables status

12.5 查看集群信息

```
127.0.0.1:7003> cluster info
cluster_state:ok
cluster_slots_assigned:16384
cluster_slots_ok:16384
cluster_slots_pfail:0
cluster_slots_fail:0
cluster_known_nodes:6
cluster_size:3
cluster_current_epoch:6
cluster_my_epoch:3
cluster_stats_messages_sent:1186
cluster_stats_messages_received:1186
```

12.6 查看集群中节点信息

```
127.0.0.1:7003> cluster nodes
713218b88321e5067fd8ad25c3bf7db88c878ccf 127.0.0.1:7003 myself,master - 0 0 3
connected 10923-16383
e7fb45e74f828b53ccd8b335f3ed587aa115b903 127.0.0.1:7001 master - 0 1498877677276
1 connected 0-5460
b1183545245b3a710a95d669d7bbcb5e09896a0 127.0.0.1:7006 slave
713218b88321e5067fd8ad25c3bf7db88c878ccf 0 1498877679294 3 connected
8879c2ed9c141de70cb7d5fcb7d690ed8a200792 127.0.0.1:7005 slave
4a312b6fc90bfee187d43588ead99d83b407c892 0 1498877678285 5 connected
4a312b6fc90bfee187d43588ead99d83b407c892 127.0.0.1:7002 master - 0 1498877674248
2 connected 5461-10922
4f8c7455574e2f0aab1e2bb341eae319ac065039 127.0.0.1:7004 slave
e7fb45e74f828b53ccd8b335f3ed587aa115b903 0 1498877680308 4 connected
```

12.7 Jedis连接集群

12.7.1 关闭防火墙

注意:如果redis重启, 需要将redis中生成的dump.rdb和nodes.conf文件删除, 然后再重启。

12.7.2 代码实现

```
<dependency>
  <groupId>redis.clients</groupId>
  <artifactId>jedis</artifactId>
  <version>2.9.0</version>
</dependency>
```

注意jedis的版本, 其他版本有可能报错:java.lang.NumberFormatException: For input string: "7002@17002"

```
public static void main(String[] args) throws IOException {
    // 创建一连接, JedisCluster对象, 在系统中是单例存在
    Set<HostAndPort> nodes = new HashSet<HostAndPort>();
    nodes.add(new HostAndPort("192.168.197.132", 7001));
    nodes.add(new HostAndPort("192.168.197.132", 7002));
    nodes.add(new HostAndPort("192.168.197.132", 7003));
    nodes.add(new HostAndPort("192.168.197.132", 7004));
    nodes.add(new HostAndPort("192.168.197.132", 7005));
    nodes.add(new HostAndPort("192.168.197.132", 7006));
    JedisCluster cluster = new JedisCluster(nodes);
}
```

```
// 执行JedisCluster对象中的方法，方法和redis指令一一对应。
cluster.set("test1", "test111");
String result = cluster.get("test1");
System.out.println(result);
//存储List数据到列表中
cluster.lpush("site-list", "java");
cluster.lpush("site-list", "c");
cluster.lpush("site-list", "mysql");
// 获取存储的数据并输出
List<String> list = cluster.lrange("site-list", 0 ,2);
for(int i=0; i<list.size(); i++) {
    System.out.println("列表项为: "+list.get(i));
}
// 程序结束时需要关闭JedisCluster对象
cluster.close();
System.out.println("集群测试成功!");
}
```

13.Redis高端面试-缓存穿透，缓存击穿，缓存雪崩问题

13.1 缓存的概念

什么是缓存?

广义的缓存就是在第一次加载某些可能会复用数据的时候，在加载数据的同时，将数据放到一个指定的地点做保存。再下次加载的时候，从这个指定地点去取数据。这里加缓存是有一个前提的，就是从这个地方取数据，比从数据源取数据要快的多。

java狭义一些的缓存，主要是指三大类

1. 虚拟机缓存 (ehcache, JBoss Cache)
2. 分布式缓存 (redis, memcache)
3. 数据库缓存

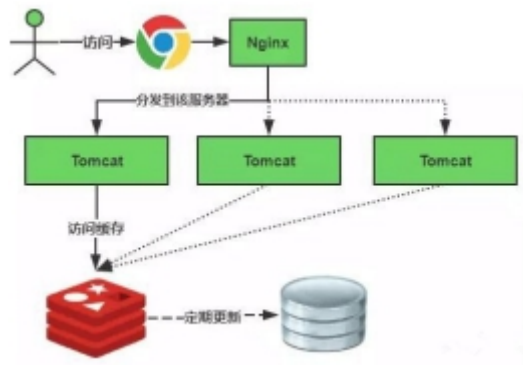
正常来说，速度由上到下依次减慢

缓存取值图:

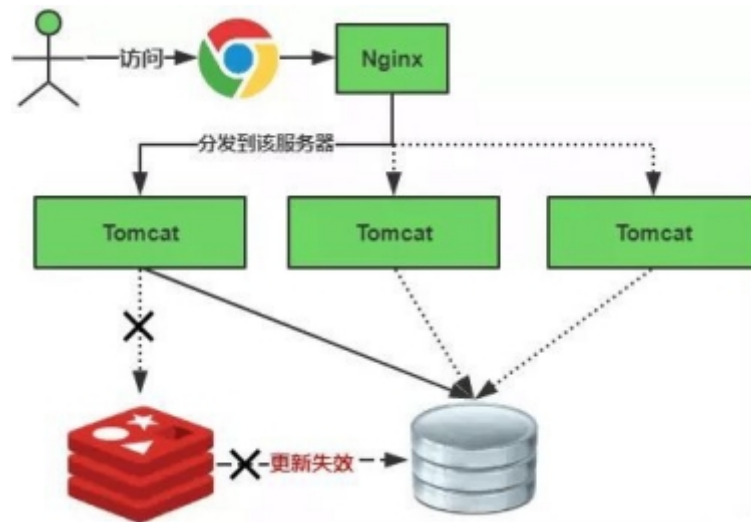
13.2 缓存雪崩

缓存雪崩产生的原因

缓存雪崩通俗简单的理解就是：由于原有缓存失效（或者数据未加载到缓存中），新缓存未到期间（缓存正常从Redis中获取，如下图）所有原本应该访问缓存的请求都去查询数据库了，而对数据库CPU和内存造成巨大压力，严重的会造成数据库宕机，造成系统的崩溃。



缓存失效的时候如下图：



解决方案：

1：在缓存失效后，通过加锁或者队列来控制读数据库写缓存的线程数量。比如对某个key只允许一个线程查询数据和写缓存，其他线程等待。虽然能够在一定的程度上缓解了数据库的压力但是与此同时又降低了系统的吞吐量。

```

public Users getByUsers(Long id) {
    // 1.先查询redis
    String key = this.getClass().getName() + "-" +
        Thread.currentThread().getStackTrace()[1].getMethodName()
        + "-id:" + id;
    String userJson = redisService.getString(key);
    if (!StringUtils.isEmpty(userJson)) {
        Users users = JSONObject.parseObject(userJson, Users.class);
        return users;
    }
    Users user = null;
    try {
        lock.lock();
        // 查询db
        user = userMapper.getUser(id);
        redisService.setSet(key, JSONObject.toJSONString(user));
    } catch (Exception e) {

    }

    } finally {
        lock.unlock(); // 释放锁
    }
    return user;
}
  
```


注意:加锁排队只是为了减轻数据库的压力,并没有提高系统吞吐量。假设在高并发下,缓存重建期间key是锁着的,这是过来1000个请求999个都在阻塞的。同样会导致用户等待超时,这是个治标不治本的方法。

2: 分析用户的行为,不同的key,设置不同的过期时间,让缓存失效的时间点尽量均匀。

13.3 缓存穿透

缓存穿透是指用户查询数据,在数据库没有,自然在缓存中也不会有。这样就导致用户查询的时候,在缓存中找不到,每次都去数据库再查询一遍,然后返回空。这样请求就绕过缓存直接查数据库,这也是经常提的缓存命中率问题。

解决方案:

1.如果查询数据库也为空,直接设置一个默认值存放到缓存,这样第二次到缓冲中获取就有值了,而不会继续访问数据库,这种办法最简单粗暴。

2.把空结果,也给缓存起来,这样下次同样的请求就可以直接返回空了,既可以避免当查询的值为空时引起的缓存穿透。同时也可以单独设置个缓存区域存储空值,对要查询的key进行预先校验,然后再放行给后面的正常缓存处理逻辑。

```
public String getByUsers2(Long id) {
    // 1.先查询redis
    String key = this.getClass().getName() + "-" +
        Thread.currentThread().getStackTrace()[1].getMethodName()+ "-id:" + id;
    String userName = redisService.getString(key);
    if (!StringUtils.isEmpty(userName)) {
        return userName;
    }
    System.out.println("#####开始发送数据库DB请求#####");
    Users user = userMapper.getUser(id);
    String value = null;
    if (user == null) {
        // 标识为null
        value = "";
    } else {
        value = user.getName();
    }
    redisService.setString(key, value);
    return value;
}
```

注意:再给对应的ip存放真值的时候,需要先清除对应的之前的空缓存。

13.4 缓存击穿

对于一些设置了过期时间的key,如果这些key可能会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,这个和缓存雪崩的区别在于这里针对某一key缓存,前者则是很多key。

热点key:

某个key访问非常频繁,当key失效的时候有大量线程来构建缓存,导致负载增加,系统崩溃。

解决办法:

①使用锁,单机用synchronized,lock等,分布式用分布式锁。

②缓存过期时间不设置,而是设置在key对应的value里。如果检测到存的时间超过过期时间则异步更新缓存。

14.Redis高端面试-分布式锁

14.1 使用分布式锁要满足的几个条件：

1. 系统是一个分布式系统（关键是分布式，单机的可以使用ReentrantLock或者synchronized代码块来实现）
2. 共享资源（各个系统访问同一个资源，资源的载体可能是传统关系型数据库或者NoSQL）
3. 同步访问（即有很多个进程同时访问同一个共享资源。）

14.2 什么是分布式锁？

线程锁：主要用来给方法、代码块加锁。当某个方法或代码使用锁，在同一时刻仅有一个线程执行该方法或该代码段。线程锁只在同一JVM中有效果，因为线程锁的实现在根本上是依靠线程之间共享内存实现的，比如synchronized是共享对象头，显示锁Lock是共享某个变量（state）。

进程锁：为了控制同一操作系统中多个进程访问某个共享资源，因为进程具有独立性，各个进程无法访问其他进程的资源，因此无法通过synchronized等线程锁实现进程锁。

分布式锁：当多个进程不在同一个系统中，用分布式锁控制多个进程对资源的访问。

14.3 应用的场景

线程间并发问题和进程间并发问题都是可以通过分布式锁解决的，但是强烈不建议这样做！因为采用分布式锁解决这些小问题是非常消耗资源的！分布式锁应该用来解决分布式情况下的多进程并发问题才是最合适的。

有这样一个情境，线程A和线程B都共享某个变量X。

如果是单机情况下（单JVM），线程之间共享内存，只要使用线程锁就可以解决并发问题。

如果是分布式情况下（多JVM），线程A和线程B很可能不是在同一JVM中，这样线程锁就无法起作用了，这时候就要用到分布式锁来解决。

分布式锁可以基于很多种方式实现，比如zookeeper、redis...。不管哪种方式，他的基本原理是不变的：用一个状态值表示锁，对锁的占用和释放通过状态值来标识。

这里主要讲如何用redis实现分布式锁。

14.4 使用redis的setNX命令实现分布式锁

实现的原理

Redis为单进程单线程模式，采用队列模式将并发访问变成串行访问，且多客户端对Redis的连接并不存在竞争关系。redis的SETNX命令可以方便的实现分布式锁。

2、基本命令解析

1) setNX (SET if Not eXists)

语法：

```
SETNX key value
```

将 key 的值设为 value，当且仅当 key 不存在。

若给定的 key 已经存在，则 SETNX 不做任何动作。

SETNX 是『SET if Not eXists』（如果不存在，则 SET）的简写

返回值：

设置成功，返回 1。

设置失败，返回 0。

例子：

```
redis> EXISTS job                # job 不存在
(integer) 0

redis> SETNX job "programmer"    # job 设置成功
(integer) 1

redis> SETNX job "code-farmer"   # 尝试覆盖 job ，失败
(integer) 0

redis> GET job                  # 没有被覆盖
"programmer"
```

所以我们使用执行下面的命令SETNX可以用作加锁原语(locking primitive)。比如说，要对关键字(key) `foo` 加锁，客户端可以尝试以下方式：

```
SETNX lock.foo <current Unix time + lock timeout + 1>
```

如果 SETNX返回 `1`，说明客户端已经获得了锁，SETNX将键 `lock.foo` 的值设置为锁的超时时间（当前时间 + 锁的有效时间）。之后客户端可以通过 `DEL lock.foo` 来释放锁。

如果 SETNX返回 `0`，说明 `key` 已经被其他客户端上锁了。如果锁是非阻塞(non blocking lock)的，我们可以选择返回调用，或者进入一个重试循环，直到成功获得锁或重试超时(timeout)。

2) getSET

先获取key对应的value值。若不存在则返回nil，然后将旧的value更新为新的value。

语法：

```
GETSET key value
```

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

当 key 存在但不是字符串类型时，返回一个错误。

返回值：

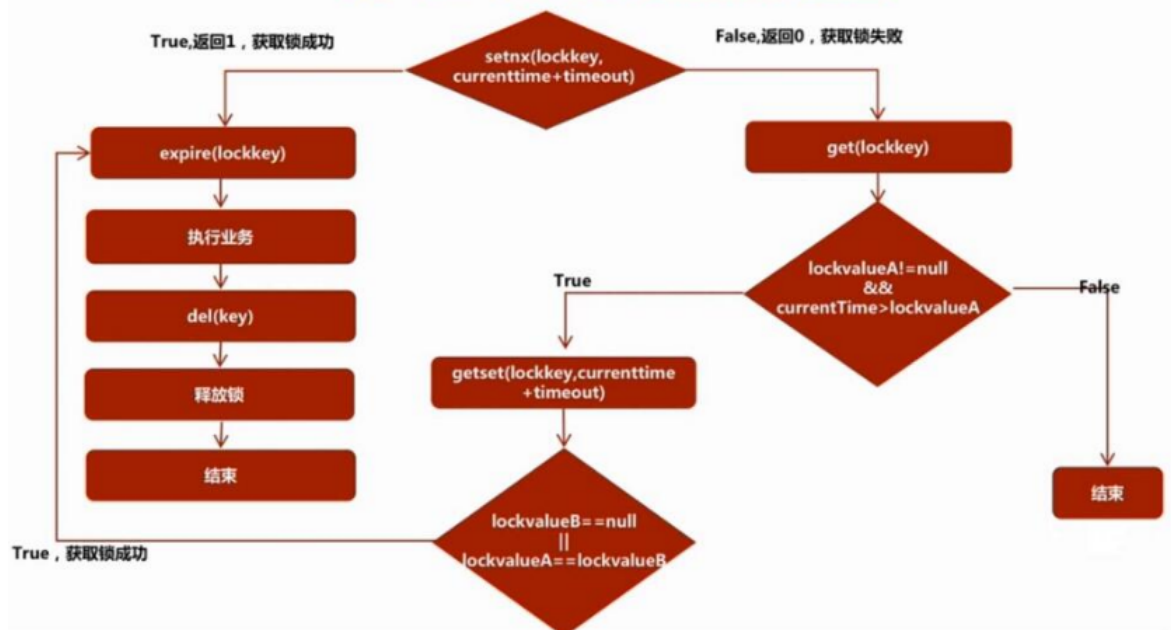
返回给定 key 的旧值[之前的值]。

当 key 没有旧值时，也即是，key 不存在时，返回 nil。

注意的关键点：(回答面试的核心点)

- 1、同一时刻只能有一个进程获取到锁。setnx
- 2、释放锁：锁信息必须是会过期超时的，不能让一个线程长期占有一个锁而导致死锁；
(最简单的方式就是del，如果在删除之前死锁了。)

Redis分布式锁优化版流程图



ex:

53秒设置--58秒到期

当前时间为56秒，没有过期

当前时间为59秒，过期。（当前时间大于设置的时间）

死锁情况是在判断超时后，直接操作业务，设置过期时间，执行业务，然后删除释放锁。其他进程再次通过setnx来抢锁。

解决死锁

上面的锁定逻辑有一个问题：**如果一个持有锁的客户端失败或崩溃了不能释放锁，该怎么解决？**

我们可以通过锁的键对应的时间戳来判断这种情况是否发生了，如果当前的时间已经大于lock.foo的值，说明该锁已失效，可以被重新使用。

发生这种情况时，可不能简单的通过DEL来删除锁，然后再SETNX一次（讲道理，**删除锁的操作应该是锁拥有者执行的，这里只需要等它超时即可**），当多个客户端检测到锁超时后都会尝试去释放它，这里就可能出现一个竞态条件,让我们模拟一下这个场景：

C0操作超时了，但它还持有着锁，C1和C2读取lock.foo检查时间戳，先后发现超时了。C1 发送DEL lock.foo C1 发送SETNX lock.foo 并且成功了。C2 发送DEL lock.foo C2 发送SETNX lock.foo 并且成功了。这样一来，C1，C2都拿到了锁！问题大了！

幸好这种问题是可以避免的，让我们来看看C3这个客户端是怎样做的：

C3发送SETNX lock.foo 想要获得锁，由于C0还持有锁，所以Redis返回给C3一个0 C3发送GET lock.foo 以检查锁是否超时了，

如果没超时，则等待或重试。反之，如果已超时，C3通过下面的操作来尝试获得锁：GETSET lock.foo <current Unix time + lock timeout + 1> 通过GETSET，C3拿到的时间戳如果仍然是超时的，那就说明，C3如愿以偿拿到锁了。如果在C3之前，有个叫C4的客户端比C3快一步执行了上面的操作，那么C3拿到的时间戳是个未超时的值，这时，C3没有如期获得锁，需要再次等待或重试。留意一下，尽管C3没拿到锁，但它改写了C4设置的锁的超时值，不过这一点非常微小的误差带来的影响可以忽略不计。

注意：为了让分布式锁的算法更稳健些，持有锁的客户端在解锁之前应该再检查一次自己的锁是否已经超时，再去做DEL操作，因为可能客户端因为某个耗时的操作而挂起，操作完的时候锁因为超时已经被别人获得，这时就不必解锁了。

伪代码：

```
public static boolean lock(String lockName) {
    Jedis jedis = RedisPool.getJedis();
    //lockName可以为共享变量名，也可以为方法名，主要是用于模拟锁信息
    System.out.println(Thread.currentThread() + "开始尝试加锁！");
    Long result = jedis.setnx(lockName,
String.valueOf(System.currentTimeMillis() + 5000));
    if (result != null && result.intValue() == 1){
        System.out.println(Thread.currentThread() + "加锁成功！");
        jedis.expire(lockName, 5);
        System.out.println(Thread.currentThread() + "执行业务逻辑！");
        jedis.del(lockName);
        return true;
    } else { //判断是否死锁
        String lockValueA = jedis.get(lockName);
        //得到锁的过期时间，判断小于当前时间，说明已超时但是没释放锁，通过下面的操作来尝试获得锁。下面逻辑防止死锁[已经过期但是没有释放锁的情况]
        if (lockValueA != null && Long.parseLong(lockValueA) <
System.currentTimeMillis()){
            String lockValueB = jedis.getSet(lockName,
String.valueOf(System.currentTimeMillis() + 5000));
            //这里返回的值是旧值，如果有的话。之前没有值就返回null,设置的是新超时。
            if (lockValueB == null || lockValueB.equals(lockValueA)){
                System.out.println(Thread.currentThread() + "加锁成功！");
                jedis.expire(lockName, 5);
                System.out.println(Thread.currentThread() + "执行业务逻辑！");
                jedis.del(lockName);
                return true;
            } else {
                return false;
            }
        } else {
            return false;
        }
    }
}
```