# Redis

## 简介

**Redis 是一个高性能的key-value数据库。** redis的出现，很大程度补偿了[**memcached**](http://baike.baidu.com/view/794242.htm)这类key/value存储的不足，在部分场合可以对**关系数据库**起到很好的补充作用。它提供了Java，C/C++，C#，PHP，JavaScript，Perl，Object-C，**Python**，Ruby，Erlang等客户端，使用很方便。

redis是一个key-value[存储系统](http://baike.baidu.com/view/51839.htm)。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](http://baike.baidu.com/view/549479.htm))、set(集合)、zset(sorted set --有序集合)和hash（哈希类型）。这些[数据类型](http://baike.baidu.com/view/675645.htm)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。

## 官网

<https://redis.io/download>

Redis 版本号采用标准惯例：主版本号.副版本号.补丁级别,一个副版本号就标记为一个标准发行版本，例如 1.2，2.0，2.2，2.4，2.6，2.8，奇数的副版本号用来表示非标准版本,例如2.9.x发行版本是Redis 3.0标准版本的非标准发行版本。

## 下载

所有版本的下载地址：http://download.redis.io/releases/

先安装下载工具：wget

yum install –y wget

wget http://download.redis.io/releases/redis-3.2.11.tar.gz

## 解压缩

tar –zxvf redis-3.2.10.tar.gz

## 编译

**注意：如果不知道，可以查看解压缩出来的readme.md文档!!**

前提条件：需要gcc编译器

yum install –y gcc-c++

make

如果报错,加参数再执行make

make MALLOC=libc

## 安装

可以指定安装的路径：

make install PREFIX=/usr/local/redis

## 测试（可选）

make test

|  |
| --- |
| Server is able to generate a stack trace on selected systems in tests/integration/logging.tcl expected stack trace not found into log file |

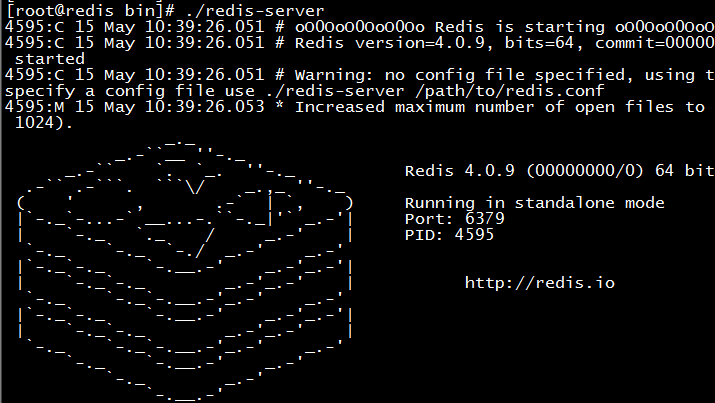
|  |
| --- |
| 我是在虚拟机中跑的make test，在github上有一个说明  For timing issues, one test isn't very representative. Did you try running them 5-10 times? Is there anything unusual about your machine (very small memory, very slow, shared, overloaded, etc)? Some of the tests are based on timing, so if the machine can't deliver results in time then tests can't complete properly. (you can manually edit some of the tests to increase the timeout waiting)  大意是说有些测试点在配置比较低的机器上会因为超时而过不了，我在虚拟机上跑的，所以很有可能是这个问题。 |

## 启动服务

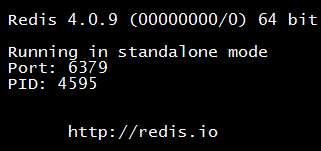
执行这个文件就可以！



./redis-server



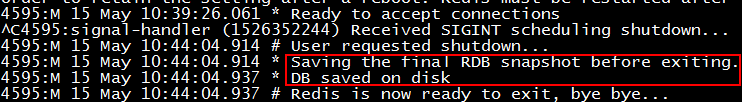
**以下信息请留意：**

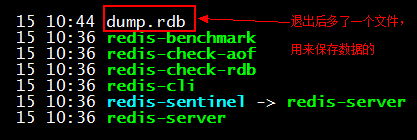


默认不是守护进程，会占用当前屏幕！！

**退出：**

ctrl+c





## 指定配置文件启动服务

举例：以守护进程方式启动(**daemon**)

需要指定一个配置文件

可以复制解压缩文件里的redis.conf文件（/root/redis-3.2.10/redis.conf）

创建一个目录保存配置文件

mkdir /usr/local/redis/conf

复制到创建的目录下

cp /root/redis-3.2.9/redis.conf /usr/local/redis/conf/

进入该目录：

cd /usr/local/redis/conf/

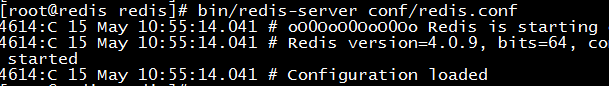
修改redis.conf文件

vi redis.conf



启动服务为守护进程

./redis-server ../conf/redis.conf



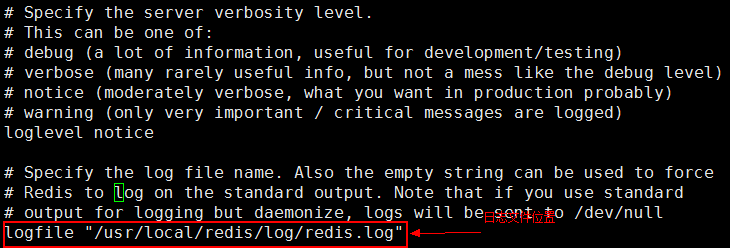
查看进程：

|  |
| --- |
| [root@redis bin]# ps aux | grep redis |



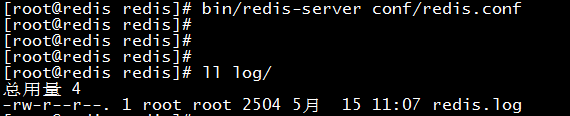
如果使用后台守护进程启动，一般需要制定日志文件！！

修改配置文件：**vi redis.conf**



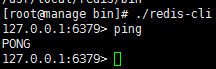
测试日志文件：

重启服务，查看日志



## 使用自带的客户端连接服务

./redis-cli



退出客户端：

quit

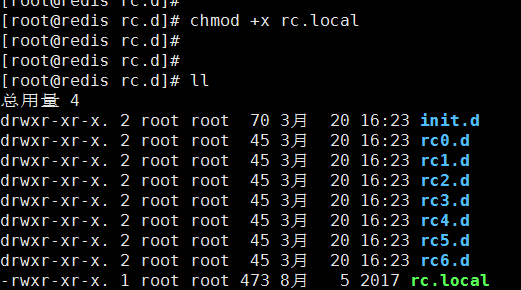
**关闭服务**：

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

## 开机自启动（方法一）

添加执行权限



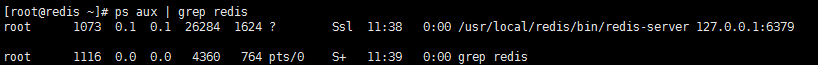


vi /etc/rc.local

添加一行：

/usr/local/redis/bin/redis-server /usr/local/redis/conf/redis.conf

可以重启系统： reboot测试是否能开机自启动！！



**开机自启动（方法2，选学）**

**注意2种启动方法只能选择一种！！**

|  |
| --- |
| 将Redis做成一个服务 service    **1.复制redis init脚本到/etc/rc.d/init.d目录**  按以上步骤默认安装Redis时，其服务脚本位于：  /root/redis-3.2.9/utils/redis\_init\_script  必须将其复制到/etc/rc.d/init.d的目录下，同时易名为redis：  cp /root/redis-3.2.9/utils/redis\_init\_script /etc/rc.d/init.d/redis    **2.将redis配置文件拷贝到/usr/local/redis/conf/目录下：**  cp /root/redis-3.2.9/redis.conf  /usr/local/redis/conf/6379.conf  这样下面3中redis服务脚本需要指定的CONF就存在了。默认情况下，Redis未启用认证，可以通过开启6379.conf的requirepass 指定一个验证密码。      **3.更改redis的服务脚本**  vi  /etc/rc.d/init.d/redis    #!/bin/sh  **#chkconfig: 2345 80 90**  # Simple Redis init.d script conceived to work on [**Linux**](http://lib.csdn.net/base/linux) systems  # as it does use of the /proc filesystem.    REDISPORT=6379  **EXEC=/usr/local/redis/bin/redis-server**  **CLIEXEC=/usr/local/redis/bin/redis-cli**  **CONF="/usr/local/redis/conf/6379.conf"**    case "$1" in      start)          if [ -f $PIDFILE]          then                  echo"$PIDFILE exists, process is already running or crashed"          else                  echo"Starting Redis server..."  **$EXEC $CONF &**          fi          ;;      stop)          if [ ! -f$PIDFILE ]          then                  echo"$PIDFILE does not exist, process is not running"          else                  PID=$(cat$PIDFILE)                  echo"Stopping ..."                  $CLIEXEC-p $REDISPORT shutdown                  while [-x /proc/${PID} ]                  do                      echo"Waiting for Redis to shutdown ..."                      sleep1                  done                  echo"Redis stopped"          fi          ;;      \*)          echo "Pleaseuse start or stop as first argument"          ;;  esac    **注：以上红色标准部分为修改的脚本：**  1)原文件是没有以下第2行的内容的，  引用#chkconfig: 2345 80 90  2)更改EXEC、CLIEXEC、CONF参数，设置对应的目录值，如上所示即和前面安装的一致。  3)更改redis开启的命令，以后台运行的方式执行：  $EXEC $CONF &  注意后面的那个“&”，即是将服务转到后面运行的意思，否则启动服务时，Redis服务将  占据在前台，占用了主用户界面，造成其它的命令执行不了。    **4.更改redis的服务脚本**  以上操作完成后，即可注册服务：  chkconfig --add redis  $ service redis start  Starting Redis server...  $ ps -ef | grep redis  root  1269   1  009:59 ?   00:00:00/usr/software/redis/redis-3.0.1/src/redis-server \*:6379  $ service redis stop  Stopping ...  Redis stopped  如果前面设置了rc.local请删除开机启动的那一行，否则会重复启动服务！！     5命令配置（可选）  **将Redis的命令所在目录添加到系统参数PATH中**  1)修改profile文件：  $ vi /etc/profile  在最后行添加:  export PATH="$PATH:/usr/local/redis/bin"    2)然后应用这个文件：  **$ source** /etc/profile    3)这样就可以直接调用redis-cli的命令了，如下所示：  **$**redis-cli  127.0.0.1:6379> |

## Redis客户端

现在收费了！！！！！

Redis-cli 命令行

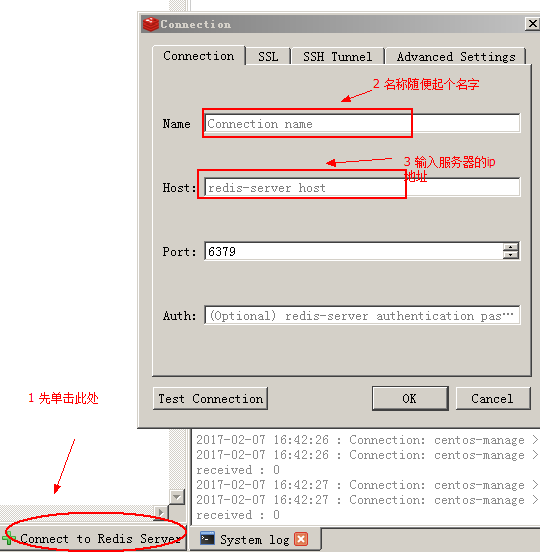
Redis-desktop-manage 图形化

官网：<https://redisdesktop.com/>

下载页面：<https://redisdesktop.com/download>

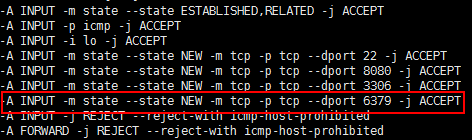
安装就是下一步即可！！

打开软件：如图操作，配置需要连接的服务器



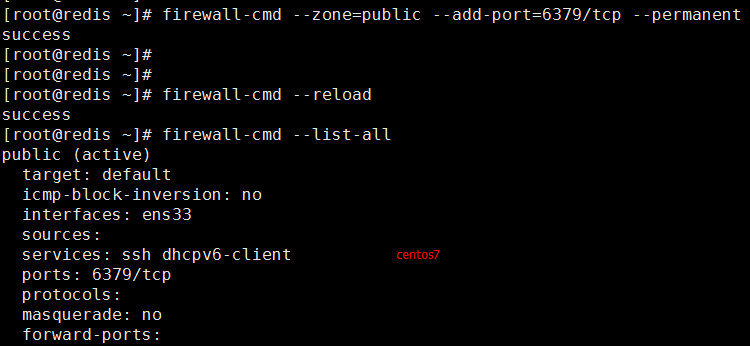
连接失败的原因：

1. 开启服务器的防火墙(6.0)

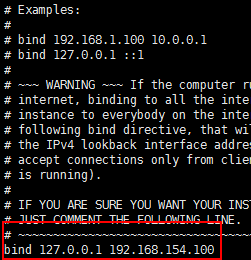


1. 重启防火墙6.0：[root@redis bin]# service iptables restart

Centos7.0开启防火墙及重启：



1. Redis的配置文件里需要绑定**实际的ip**地址(**默认不支持远程连接**)

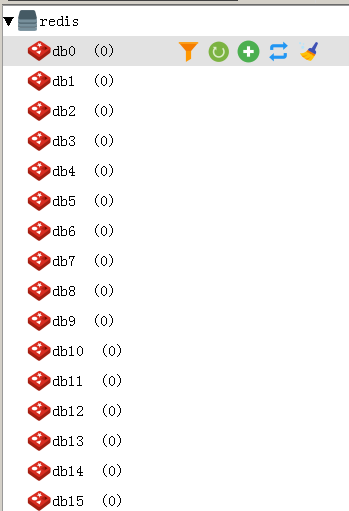


4, 重启redis服务器：

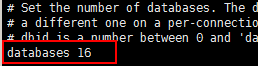
[root@redis bin]# ./redis-cli shutdown

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

连接成功后如图所示：



默认有16个数据库,在配置文件里设置



## 常用的redis命令

中文官网：<http://www.redis.cn/>

命令大全：<http://www.redis.cn/commands.html>

### String命令

Set 添加数据：

redis> SET mykey "Hello"

OK

Get查询数据：

redis> GET mykey

"Hello"

Set修改数据：

redis:0>set mykey "hello2"

OK

redis:0>get mykey

hello2

del 删除数据：

redis:0>del mykey

1

redis:0>get mykey

NULL

### Hash命令

Hset 添加数据：

redis> HSET myhash field1 "Hello"

(integer) 1

HGet查询数据：

redis> HGET myhash field1

"Hello"

HSet修改数据：

redis:0>hset myhash mykey 200

0

redis:0>hget myhash mykey

200

hdel 删除数据：

redis:0>hdel myhash mykey

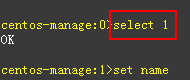
1

redis:0>hget myhash mykey

NULL

### select

切换数据库 select：



### ttl

查询指定值的生存时间

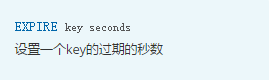
默认时间：-1，代表永不过期。

时间单位是**秒**

redis:15>ttl age

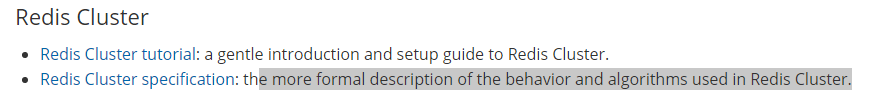
-1

### expire



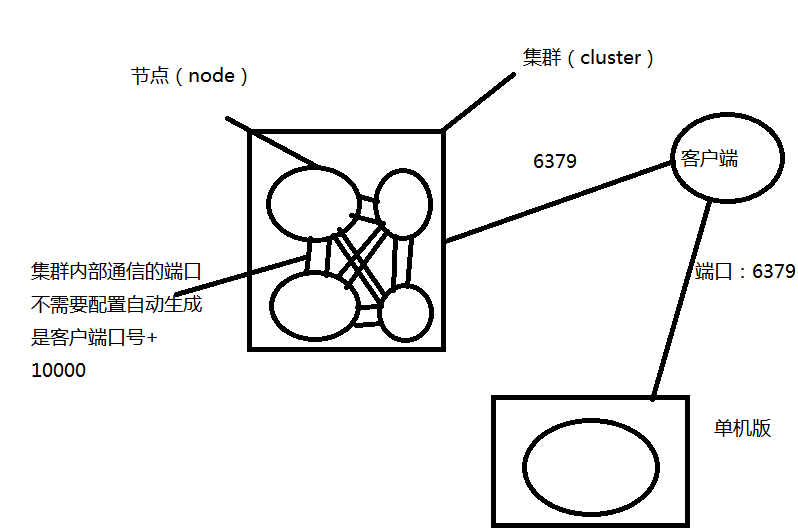
# Redis集群

Redis Cluster



集群搭建：<https://redis.io/topics/cluster-tutorial>

集群规范：<https://redis.io/topics/cluster-spec>



|  |
| --- |
| Note that for a Redis Cluster to work properly you need, for each node:   1. The normal client communication port (usually 6379) used to communicate with clients to be open to all the clients that need to reach the cluster, plus all the other cluster nodes (that use the client port for keys migrations). 2. The cluster bus port (the client port + 10000) must be reachable from all the other cluster nodes. |

数据分片（data sharding）

|  |
| --- |
| Redis Cluster data sharding Redis Cluster does not use consistent hashing, but a different form of sharding where every key is conceptually part of what we call an **hash slot**.  There are **16384** hash slots in Redis Cluster, and to compute what is the hash slot of a given key, we simply take the **CRC16** of the key modulo **16384**.  Every node in a Redis Cluster is responsible for a subset of the hash slots, so for example you may have a cluster with 3 nodes, where:   * Node A contains hash slots from 0 to 5500. * Node B contains hash slots from 5501 to 11000. * Node C contains hash slots from 11001 to 16383. |

主备服务器（master-slave）

主要作用就是备份数据，实现高可用

|  |
| --- |
| Redis Cluster master-slave model In order to remain available when a subset of master nodes are failing or are not able to communicate with the majority of nodes, Redis Cluster uses a master-slave model where every hash slot has from 1 (the master itself) to N replicas (N-1 additional slaves nodes). |

## 集群创建

备注：我们的集群时“伪”集群，因为真正的集群不可能放在同一个服务器上。

mkdir redis-cluster

**在复制前，请先删除原来单机版下的所有数据，否则后面分配插槽会失败！！**

然后复制6份redis分别放在不同编号的目录下：

cp redis /usr/local/redis-cluster/7000 -r

[root@redis local]# cp redis /usr/local/redis-cluster/7001 -r

[root@redis local]# cp redis /usr/local/redis-cluster/7002 -r

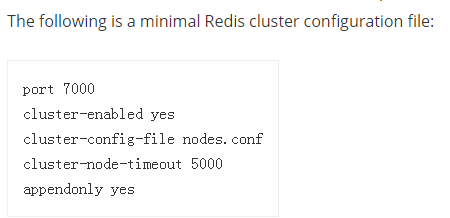
[root@redis local]# cp redis /usr/local/redis-cluster/7003 -r

[root@redis local]# cp redis /usr/local/redis-cluster/7004 -r

[root@redis local]# cp redis /usr/local/redis-cluster/7005 –r

cd /usr/local/redis-cluster/7000

修改每个配置文件：







修改以上配置文件内容！！

后面重复以上步骤配置其他节点！！

可以使用复制7000的配置文件，然后查找替换

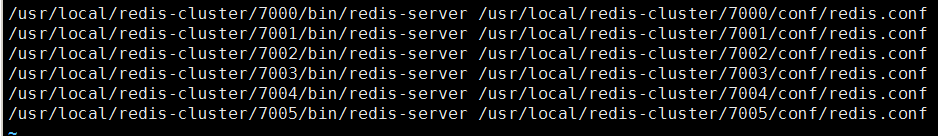


替换结果：



全部配置成功后，写个启动脚本**startup.sh**：

|  |
| --- |
| /usr/local/redis-cluster/7000/bin/redis-server /usr/local/redis-cluster/7000/conf/redis.conf  /usr/local/redis-cluster/7001/bin/redis-server /usr/local/redis-cluster/7001/conf/redis.conf  /usr/local/redis-cluster/7002/bin/redis-server /usr/local/redis-cluster/7002/conf/redis.conf  /usr/local/redis-cluster/7003/bin/redis-server /usr/local/redis-cluster/7003/conf/redis.conf  /usr/local/redis-cluster/7004/bin/redis-server /usr/local/redis-cluster/7004/conf/redis.conf  /usr/local/redis-cluster/7005/bin/redis-server /usr/local/redis-cluster/7005/conf/redis.conf |

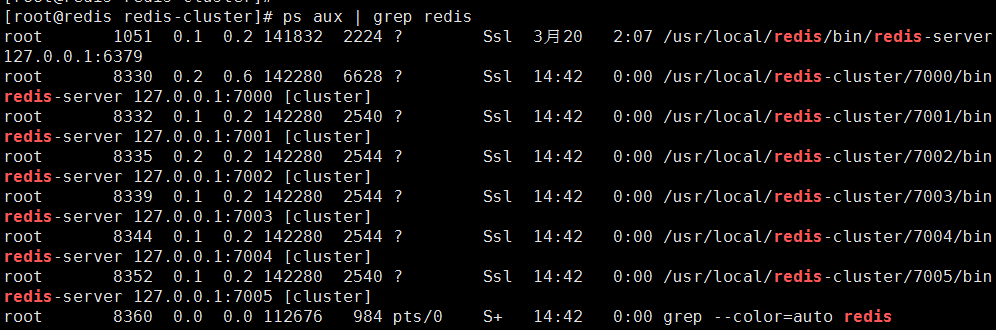


修改执行权限：



启动6个redis实例：

**./startup.sh**



下面我们要用官方提供的建立集群的脚本来创建集群：



由于此脚本是用“ruby”写的，所有要运行此脚本需要安装“ruby”运行环境：



**Yum install rubygems**

执行：**gem install redis**

提示出错：

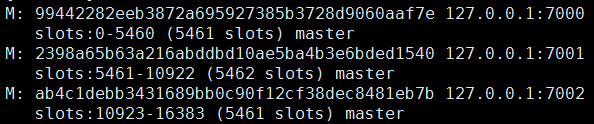
[**redis requires Ruby version >= 2.2.2问题**](http://www.cnblogs.com/carryping/p/7447823.html)

请按如下步骤解决：

|  |
| --- |
| redis requires Ruby version >= 2.2.2的报错，查了资料发现是Centos默认支持ruby到2.0.0，可gem 安装[**redis**](http://lib.csdn.net/base/redis)需要最低是2.2.2  解决办法是 先安装rvm，再把ruby版本提升至2.3.3  1.安装curl   yum install curl  2. 安装RVM  curl -L get.rvm.io | bash -s stable  如果这一步也报错：    再重新执行以上命令提示安装成功：    3. 启用rvm  source /usr/local/rvm/scripts/rvm  或者 source /etc/profile.d/rvm.sh  4. 查看rvm库中已知的ruby版本  rvm list known  5. 安装一个ruby版本  rvm install 2.3.4  6. 使用一个ruby版本  rvm use 2.3.4  7. 卸载一个已知版本  rvm remove 2.0.0  8.  查看默认版本  **ruby –version**    9. 再安装redis就可以了  **gem install redis** |

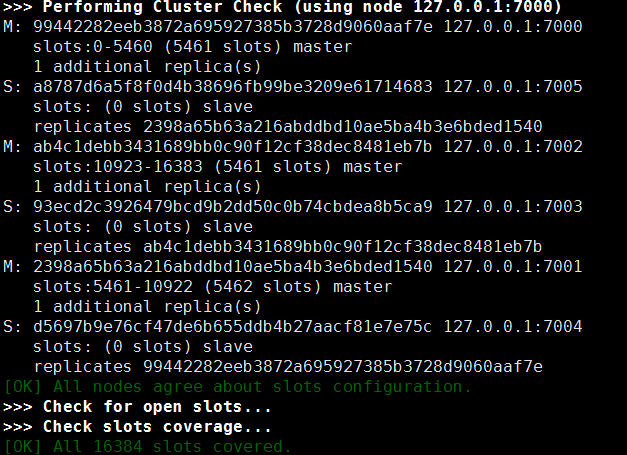
**注意点：这里如果只是做本地测试可以使用本地回环地址，如下操作**

**./redis-trib.rb create --replicas 1 127.0.0.1:7000 127.0.0.1:7001 127.0.0.1:7002 127.0.0.1:7003 127.0.0.1:7004 127.0.0.1:7005**

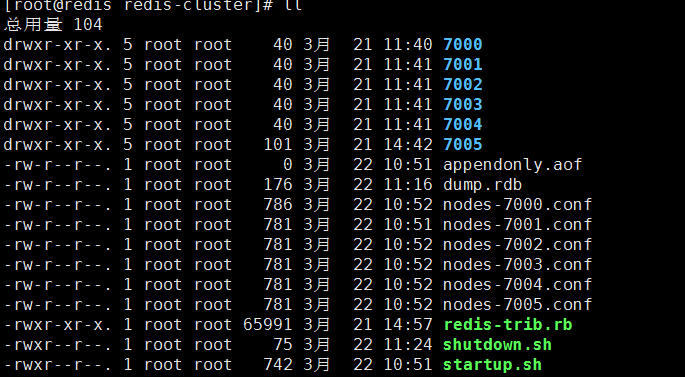


输入“yes”

分配成功：



目录结构：



测试集群：

$ redis-cli -c -p 7000

redis 127.0.0.1:7000> set foo bar

-> Redirected to slot [12182] located at 127.0.0.1:7002

OK

redis 127.0.0.1:7002> set hello world

-> Redirected to slot [866] located at 127.0.0.1:7000

OK

redis 127.0.0.1:7000> get foo

-> Redirected to slot [12182] located at 127.0.0.1:7002

"bar"

redis 127.0.0.1:7000> get hello

-> Redirected to slot [866] located at 127.0.0.1:7000

"world"

**如果要配置成远程集群访问，请在创建集群时一定使用配置的网卡地址，同时要bind住这个地址**

**最好将远程的ip配置在本地回环地址之前，或者干脆只配置前一个！！切记！！**

**否则可能客户端能连接，但是使用jedis连接时会出现连接不上的错误！！**

**./redis-trib.rb create --replicas 1 192.168.192.40:7000 192.168.192.40:7001 192.168.192.40:7002 192.168.192.40:7003 192.168.192.40:7004 192.168.192.40:7005**

**为了方便我们快速启动集群和关闭集群，可以写成相应的脚本如下：**

**如果想自动输入“yes”，可以在创建集群的命令前加上 “echo yes | ”**

**Startup.sh:**

|  |
| --- |
| **cd /usr/local/redis-cluster**  **rm -rf appendonly.aof dump.rdb nodes-\*.conf**  **/usr/local/redis-cluster/7000/bin/redis-server /usr/local/redis-cluster/7000/conf/redis.conf**  **/usr/local/redis-cluster/7001/bin/redis-server /usr/local/redis-cluster/7001/conf/redis.conf**  **/usr/local/redis-cluster/7002/bin/redis-server /usr/local/redis-cluster/7002/conf/redis.conf**  **/usr/local/redis-cluster/7003/bin/redis-server /usr/local/redis-cluster/7003/conf/redis.conf**  **/usr/local/redis-cluster/7004/bin/redis-server /usr/local/redis-cluster/7004/conf/redis.conf**  **/usr/local/redis-cluster/7005/bin/redis-server /usr/local/redis-cluster/7005/conf/redis.conf**  **echo yes | ./redis-trib.rb create --replicas 1 192.168.192.40:7000 192.168.192.40:7001 192.168.192.40:7002 192.168.192.40:7003 192.168.192.40:7004 192.168.192.40:7005** |

**Shutdown.sh:**

|  |
| --- |
| **ps aux | grep redis-cluster | grep -v grep | awk {'print $2'} | xargs kill** |

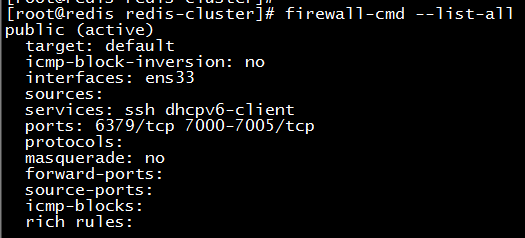
## 集群本地测试

|  |
| --- |
| $ redis-cli -c -p 7000  redis 127.0.0.1:7000> set foo bar  -> Redirected to slot [12182] located at 127.0.0.1:7002  OK  redis 127.0.0.1:7002> set hello world  -> Redirected to slot [866] located at 127.0.0.1:7000  OK  redis 127.0.0.1:7000> get foo  -> Redirected to slot [12182] located at 127.0.0.1:7002  "bar"  redis 127.0.0.1:7000> get hello  -> Redirected to slot [866] located at 127.0.0.1:7000  "world" |

## 远程客户端测试

首先记得开防火墙：

**firewall-cmd --zone=public --add-port=7000-7005/tcp –permanent**



# Jedis

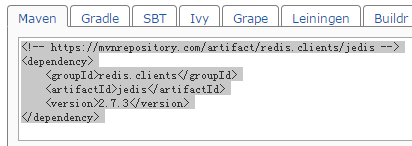
**jdbc**是用来连接**关系型**数据库，非关系型，没有统一的接口，只能寻找各个数据库自己开发的java版本的类库（jar包）！！

源码地址：https://github.com/xetorthio/jedis

Jedis is a blazingly small and sane Redis java client.

**Java程序连接redis的jar包！！**

<https://mvnrepository.com/artifact/redis.clients/jedis>





创建一个测试项目（maven项目）

添加依赖：

|  |
| --- |
| <properties>  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>  <maven.test.failure.ignore>true</maven.test.failure.ignore>  <maven.test.skip>true</maven.test.skip>  <jdk.version>1.7</jdk.version>  </properties>  <dependencies>  <!-- jedis -->  <dependency>  <groupId>redis.clients</groupId>  <artifactId>jedis</artifactId>  <version>2.9.0</version>  </dependency>  <!-- junit -->  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>4.12</version>  <scope>test</scope>  </dependency>  </dependencies>  <build>  <plugins>  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-compiler-plugin</artifactId>  <version>3.5.1</version>  <configuration>  <encoding>UTF-8</encoding>  <source>${jdk.version}</source>  <target>${jdk.version}</target>  </configuration>  </plugin>  </plugins>  </build> |

## 单机版测试

创建测试类JedisTest：

|  |
| --- |
| package edu.cjl.jedis;  import static org.junit.Assert.\*;  import org.junit.Test;  import redis.clients.jedis.Jedis;  import redis.clients.jedis.JedisPool;  public class JedisTest {  /\*\*  \* string 类型的操作  \*/  @Test  public void testSet() {  Jedis jedis = new Jedis("192.168.37.40",6379);  jedis.set("key1", "value1");  String value = jedis.get("key1");  jedis.close();  assertEquals("value1", value);  }    @Test  public void testDel() {  Jedis jedis = new Jedis("192.168.37.40",6379);  jedis.del("key1");  String value = jedis.get("key1");  jedis.close();  assertEquals(null, value);  }    /\*\*  \* hash 操作  \*/  @Test  public void testHset() {  Jedis jedis = new Jedis("192.168.37.40",6379);  jedis.hset("myhash1","key1", "value1");  String value = jedis.hget("myhash1","key1");  jedis.close();  assertEquals("value1", value);  }    @Test  public void testHdel() {  Jedis jedis = new Jedis("192.168.37.40",6379);  jedis.hdel("myhash1","key1");  String value = jedis.hget("myhash1","key1");  jedis.close();  assertEquals(null, value);  }    /\*\*  \* ttl, expire  \*/  @Test  public void testTtl() {  Jedis jedis = new Jedis("192.168.37.40",6379);  jedis.set("key1", "value1");  Long ttl = jedis.ttl("key1");  jedis.close();  assertEquals(-1, ttl.longValue());  }    @Test  public void testExpire() {  Jedis jedis = new Jedis("192.168.37.40",6379);  jedis.set("key1", "value2");  jedis.expire("key1", 30);  Long ttl = jedis.ttl("key1");  jedis.close();  assertEquals(30, ttl.longValue());  }    /\*\*  \* 生产环境需要使用连接池 jedispool  \*/  @Test  public void testSetWithPool() {  JedisPool pool = new JedisPool("192.168.37.40",6379);  Jedis jedis = pool.getResource();  jedis.set("key1", "pool");  String value = jedis.get("key1");  jedis.close();  assertEquals("pool", value);  }  } |

## 测试客户端集群连接

|  |
| --- |
| package com.wg.jedis;  import static org.junit.Assert.\*;  import java.util.HashSet;  import java.util.Set;  import org.junit.BeforeClass;  import org.junit.Test;  import redis.clients.jedis.HostAndPort;  import redis.clients.jedis.JedisCluster;  public class JedisClusterTest {    private static JedisCluster jc ;    //测试运行时执行一次，一般用来初始化属性值  @BeforeClass  public static void setUpBeforeClass() throws Exception {    Set<HostAndPort> jedisClusterNodes = new HashSet<HostAndPort>();  //Jedis Cluster will attempt to discover cluster nodes automatically  jedisClusterNodes.add(new HostAndPort("192.168.192.50", 7000));  jedisClusterNodes.add(new HostAndPort("192.168.192.50", 7001));  jedisClusterNodes.add(new HostAndPort("192.168.192.50", 7002));  jedisClusterNodes.add(new HostAndPort("192.168.192.50", 7003));  jedisClusterNodes.add(new HostAndPort("192.168.192.50", 7004));  jedisClusterNodes.add(new HostAndPort("192.168.192.50", 7005));  jc = new JedisCluster(jedisClusterNodes);  }    @Test  public void testSet() {  String res = jc.set("c-test", "001");  assertEquals("OK", res);  }    @Test  public void testGet() {  String res = jc.get("c-test");  assertEquals("001", res);  }  @Test  public void testDel() {  Long res = jc.del("c-test");  assertEquals(1, res.longValue());  }  } |

# Spring-boot-redis

官网：<https://docs.spring.io/spring-boot/docs/1.5.13.RELEASE/reference/htmlsingle/#boot-features-nosql>

|  |
| --- |
| Spring Data provides additional projects that help you access a variety of NoSQL technologies including [MongoDB](https://projects.spring.io/spring-data-mongodb/), [Neo4J](https://projects.spring.io/spring-data-neo4j/), [Elasticsearch](https://github.com/spring-projects/spring-data-elasticsearch/), [Solr](https://projects.spring.io/spring-data-solr/), [Redis](https://projects.spring.io/spring-data-redis/), [Gemfire](https://projects.spring.io/spring-data-gemfire/),[Cassandra](https://projects.spring.io/spring-data-cassandra/), [Couchbase](https://projects.spring.io/spring-data-couchbase/) and [LDAP](https://projects.spring.io/spring-data-ldap/). Spring Boot provides auto-configuration for Redis, MongoDB, Neo4j, Elasticsearch, Solr Cassandra, Couchbase and LDAP; you can make use of the other projects, but you will need to configure them yourself. Refer to the appropriate reference documentation at [projects.spring.io/spring-data](https://projects.spring.io/spring-data). |

|  |
| --- |
| [Redis](http://redis.io/) is a cache, message broker and richly-featured key-value store. Spring Boot offers basic auto-configuration for the [Jedis](https://github.com/xetorthio/jedis/) client library and abstractions on top of it provided by [Spring Data Redis](https://github.com/spring-projects/spring-data-redis). There is a spring-boot-starter-data-redis ‘Starter’ for collecting the dependencies in a convenient way. |

|  |
| --- |
| You can inject an auto-configured RedisConnectionFactory, StringRedisTemplate or vanilla RedisTemplate instance as you would any other Spring Bean. By default the instance will attempt to connect to a Redis server using localhost:6379 |

Spring-data-redis: <https://docs.spring.io/spring-data/redis/docs/1.8.12.RELEASE/reference/html/#redis>

**Redis的spring boot配置类：**

|  |
| --- |
| @Configuration  @ConditionalOnClass({ JedisConnection.**class**, RedisOperations.**class**, Jedis.**class** })  @EnableConfigurationProperties(RedisProperties.**class**)  **public** **class** RedisAutoConfiguration |

|  |
| --- |
| @ConfigurationProperties(prefix = "spring.redis")  **public** **class** RedisProperties {  /\*\*  \* Database index used by the connection factory.  \*/  **private** **int** database = 0;  /\*\*  \* Redis url, which will overrule host, port and password if set.  \*/  **private** String url; |

# 使用redis缓存商品分类数据

## 流程分析

商品分类查询

代码

先查询缓存

存在

没有

数据库

将查询结果添加至缓存

查询

返回结果

**缓存预热？？？？？**

## Redis配置

|  |
| --- |
| #datasource  spring:  datasource:  type: com.alibaba.druid.pool.DruidDataSource  url: jdbc:mysql://localhost:3306/sc-admin?characterEncoding=utf-8  username: root  password: 111111  driver-class-name: com.mysql.jdbc.Driver  redis:  cluster:  nodes: 192.168.192.40:7000,192.168.192.40:7001,192.168.192.40:7002,192.168.192.40:7003,192.168.192.40:7004,192.168.192.40:7005  application:  name: service-goodscat  #service  eureka:  client:  serviceUrl:  defaultZone: http://localhost:8760/eureka/  server:  port: 8761 |

## Redis依赖

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-data-redis</artifactId>  </dependency> |

## 接口定义

|  |
| --- |
| package com.sc.service;  import java.util.List;  import com.sc.pojo.GoodsCat;  public interface GoodsCatService {  String REDIS\_KEY\_GOODSCAT\_TOP = "goodscat:top";    String REDIS\_KEY\_GOODSCAT\_ALL = "goodscat:all";    List<GoodsCat> getGoodsCatByPid(int pid);    List<GoodsCat> getAllByPid(int pid);  } |

|  |
| --- |
| **package** com.sc.redis;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.data.redis.core.StringRedisTemplate;  **import** org.springframework.stereotype.Repository;  @Repository  **public** **class** GoodsCatCacheImpl **implements** GoodsCatCache {  @Autowired  **private** StringRedisTemplate template;    @Override  **public** String get(String key) {  **return** template.opsForValue().get(key);  }  @Override  **public** **void** set(String key, String value) {  template.opsForValue().set(key, value);  }    } |

## 业务类

|  |
| --- |
| package com.sc.service;  import java.util.List;  import org.slf4j.Logger;  import org.slf4j.LoggerFactory;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.stereotype.Service;  import com.sc.mapper.GoodsCatMapper;  import com.sc.pojo.GoodsCat;  import com.sc.redis.GoodsCatCache;  import com.sc.utils.JsonUtils;  @Service  public class GoodsCatServiceImpl implements GoodsCatService {  private static final Logger logger = LoggerFactory.getLogger(GoodsCatServiceImpl.class);    @Autowired  private GoodsCatMapper goodsCatMapper;  @Autowired  private GoodsCatCache goodsCatCache;    @Override  public List<GoodsCat> getGoodsCatByPid(int pid) {  **String top = null;**  **try {**  **top = goodsCatCache.get(GoodsCatService.REDIS\_KEY\_GOODSCAT\_TOP);**  **} catch (Exception e) {**  **logger.error("redis get goodscat top cache exception", e);**  **}**    **if(top!=null) {**  **return JsonUtils.jsonToList(top, GoodsCat.class);**  **}**      List<GoodsCat> cats = goodsCatMapper.selectGoodsCatByPid(pid);  这段代码是有问题的，因为网络或者其他的原因导致连接超时，怎么办??主要是同步调用，改为异步？？mq消息队列    解耦合  **try {**  **goodsCatCache.set(REDIS\_KEY\_GOODSCAT\_TOP, JsonUtils.objectToJson(cats));**  **} catch (Exception e) {**  **logger.error("redis add goodscat top cache exception", e);**  **}**    return cats;  }    @Override  public List<GoodsCat> getAllByPid(int pid) {    **String all = null;**  **try {**  **all = goodsCatCache.get(GoodsCatService.REDIS\_KEY\_GOODSCAT\_ALL);**  **} catch (Exception e) {**  **logger.error("redis get goodscat all cache exception", e);**  **}**    **if(all!=null) {**  **return JsonUtils.jsonToList(all, GoodsCat.class);**  **}**    List<GoodsCat> parents = goodsCatMapper.selectGoodsCatByPid(pid);  for(GoodsCat cat : parents) {  if(cat.getIsParent()) {  List<GoodsCat> childs = getAllByPid(cat.getId());  cat.setChildren(childs);  }  }    **try {**  **goodsCatCache.set(REDIS\_KEY\_GOODSCAT\_ALL, JsonUtils.objectToJson(parents));**  **} catch (Exception e) {**  **logger.error("redis add goodscat all cache exception", e);**  **}**    return parents;  }  } |

## 测试类

|  |
| --- |
| package com.sc.service;  import java.util.List;  import org.junit.Test;  import org.junit.runner.RunWith;  import org.slf4j.Logger;  import org.slf4j.LoggerFactory;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.boot.test.context.SpringBootTest;  import org.springframework.test.context.junit4.SpringRunner;  import com.sc.GoodsCatMain;  import com.sc.pojo.GoodsCat;  @RunWith(SpringRunner.class)  @SpringBootTest(classes = GoodsCatMain.class)  public class GoodsCatServiceTest {  private final static Logger logger = LoggerFactory.getLogger(GoodsCatServiceTest.class);    @Autowired  private GoodsCatService gcs;    @Test  public void testGetAllByPid() {  List<GoodsCat> all = gcs.getAllByPid(0);  logger.info(all.toString());  }    @Test  public void testGetTopByPid() {  List<GoodsCat> top = gcs.getGoodsCatByPid(0);  logger.info(top.toString());  }    } |

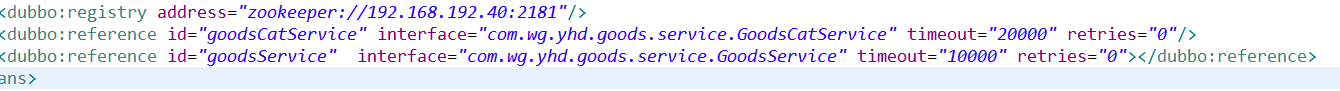
## 测试集群缓存崩溃时业务是否正常

**注意先试一试重启服务，一定要重启，服务端，客户端都重启一下，比较保险！！**

由于dubbo客户端调用服务端有默认的过期时间，比较短，过期后会抛异常，导致看上去页面崩溃是由缓存引起的，是代码问题，然而并不是，是默认情况下，redis请求超时时间比较长，导致调用服务的客户端抛出超时异常！！

解决方法：

修改默认超时时间，



# 问题？？

如何实现缓存同步？？你能想到什么方案？？考虑生成环境的复杂性？？

由于商品分类这个业务很少变更，所以此处不太适合做缓存同步，后面我们选择同步商品缓存来演示。

**但是这里请先思考如何在分布式环境下同步缓存！！！（主要是考虑数据一致性的强度要求。）**

# Redis集群有坑

<http://www.mamicode.com/info-detail-1523178.html>

# 持久化方式（面试会问）

#### 一、RDB（snapshot）概述

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**

#### 二、AOF概述

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。**

**三、RDB-AOF**

**4.0开始允许使用RDB-AOF混合持久化的方式结合了两者的优点通过aof-use-rdb-preamble配置项可以打开混合开关。**

#### 三.总结

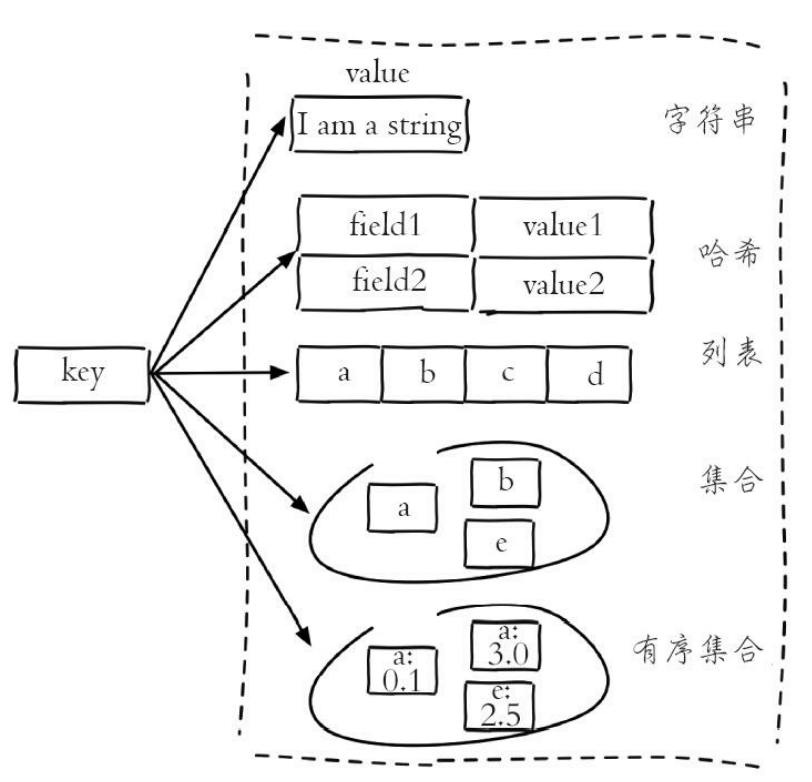
AOF和RDB各有优缺点，这是有它们各自的特点所决定：

* 1) AOF更加安全，可以将数据更加及时的同步到文件中，但是AOF需要较多的磁盘IO开支，AOF文件尺寸较大，文件内容恢复数度相对较慢。   
  2) snapshot，安全性较差，它是“正常时期”数据备份以及master-slave数据同步的最佳手段，文件尺寸较小，恢复数度较快。

可以通过配置文件来指定它们中的一种，或者同时使用它们(不建议同时使用)，或者全部禁用，在架构良好的环境中，master通常使用AOF，slave使用snapshot，主要原因是master需要首先确保数据完整性，它作为数据备份的第一选择；slave提供只读服务(目前slave只能提供读取服务)，它的主要目的就是快速响应客户端read请求；但是如果你的redis运行在网络稳定性差/物理环境糟糕情况下，建议你master和slave均采取AOF，这个在master和slave角色切换时，可以减少“人工数据备份”/“人工引导数据恢复”的时间成本；如果你的环境一切非常良好，且服务需要接收密集性的write操作，那么建议master采取snapshot，而slave采用AOF。

# Redis 基础数据结构

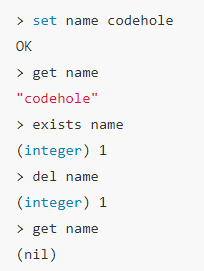
Redis 有 5 种基础数据结构，分别为：string (字符串)、list (列表)、set (集合)、hash (哈希) 和 zset (有序集合)。



## string (字符串)

字符串 string 是 Redis 最简单的数据结构。Redis 所有的数据结构都是以唯一的 key 字符串作为名称，然后通过这个唯一 key 值来获取相应的 value 数据。不同类型的数据结构的差异就在于 value 的结构不一样。字符串结构使用非常广泛，一个常见的用途就是缓存用户信息。我们将用户信息结构体使用 JSON 序列化成字符串，然后将序列化后的字符串塞进 Redis 来缓存。同样，取用户信息会经过一次反序列化的过程。

**键值对**



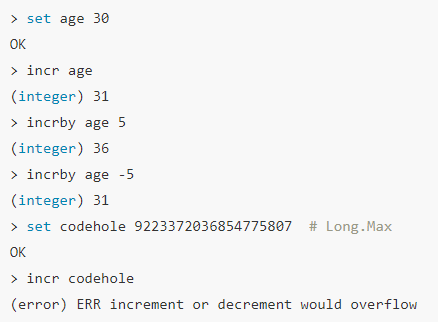
**批量键值对：**可以批量对多个字符串进行读写，节省网络耗时开销



**过期和 set 命令扩展：**可以对 key 设置过期时间，到点自动删除，这个功能常用来控制缓存的失效时间



**原子计数：**如果 value 值是一个整数，还可以对它进行自增操作。自增是有范围的，它的范围是 signed long 的最大最小值，超过了这个值，Redis 会报错



## list (列表)

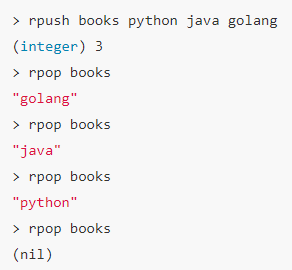
Redis 的列表相当于 Java 语言里面的 LinkedList，注意它是链表而不是数组。这意味着 list 的插入和删除操作非常快，时间复杂度为 O(1)，但是索引定位很慢，时间复杂度为 O(n)。 当列表弹出了最后一个元素之后，该数据结构自动被删除，内存被回收。

Redis 的列表结构常用来做**异步队列**使用。将需要延后处理的任务结构体序列化成字符串塞进 Redis 的列表，另一个线程从这个列表中轮询数据进行处理。

**右边进左边出：队列**



**右边进右边出：栈**



## hash (字典)

Redis 的字典相当于 Java 语言里面的 HashMap，它是无序字典。内部实现结构上同 Java 的 HashMap 也是一致的，同样的数组 + 链表二维结构。第一维 hash 的数组位置碰撞时，就会将碰撞的元素使用链表串接起来。

hash 结构也可以用来存储用户信息，不同于字符串一次性需要全部序列化整个对象，hash 可以对 用户结构中的每个字段单独存储。这样当我们需要获取用户信息时可以进行部分获取。而以整个字符串的形式去保存用户信息的话就只能一次性全部读取，这样就会比较浪费网络流量。 hash 也有缺点，hash 结构的存储消耗要高于单个字符串，到底该使用 hash 还是字符串，需要根据实际情况再三权衡。



## 

## set (集合)

Redis 的集合相当于 Java 语言里面的 HashSet，它内部的键值对是无序的唯一的。它的内部实现相当于一个特殊的字典，字典中所有的 value 都是一个值NULL。 当集合中最后一个元素移除之后，数据结构自动删除，内存被回收。

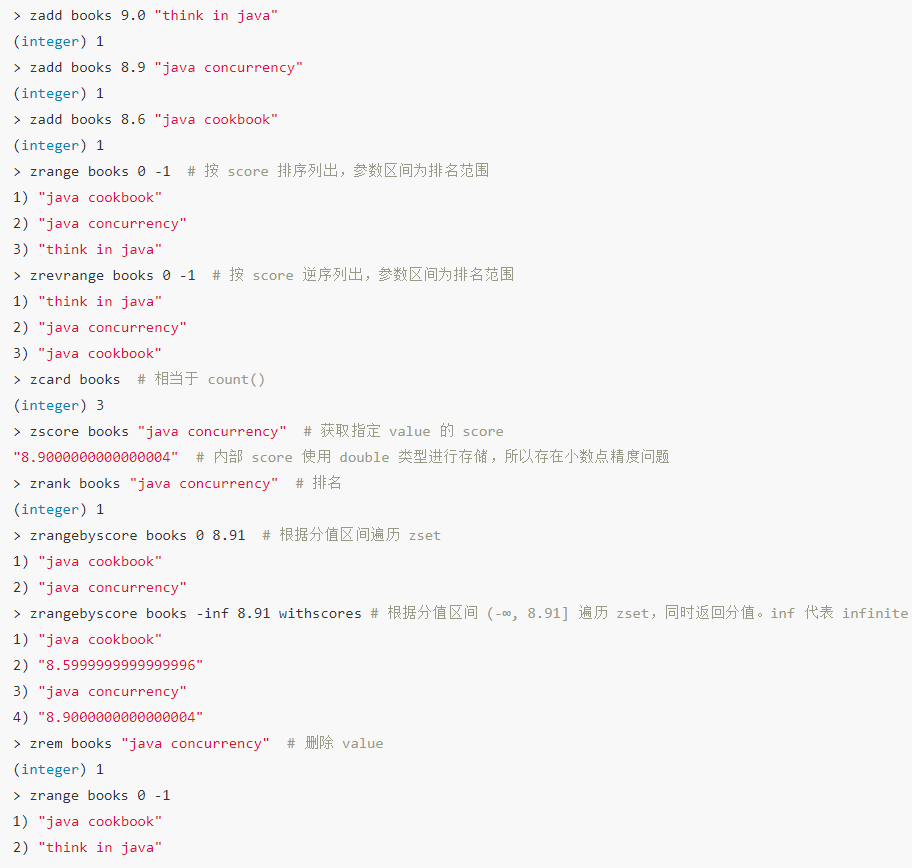


## zset (有序集合)

zset 似于 Java 的 SortedSet 和 HashMap 的结合体，一方面它是一个 set，保证了内部 value 的唯一性，另一方面它可以给每个 value 赋予一个 score，代表这个 value 的排序权重。

**zset** 可以用来存**粉丝列表**，value 值是粉丝的用户 ID，score 是关注时间。我们可以对粉丝列表按关注时间进行排序。

zset 还可以用来存储学生的成绩，value 值是学生的 ID，score 是他的考试成绩。我们可以对成绩按分数进行排序就可以得到他的名次。

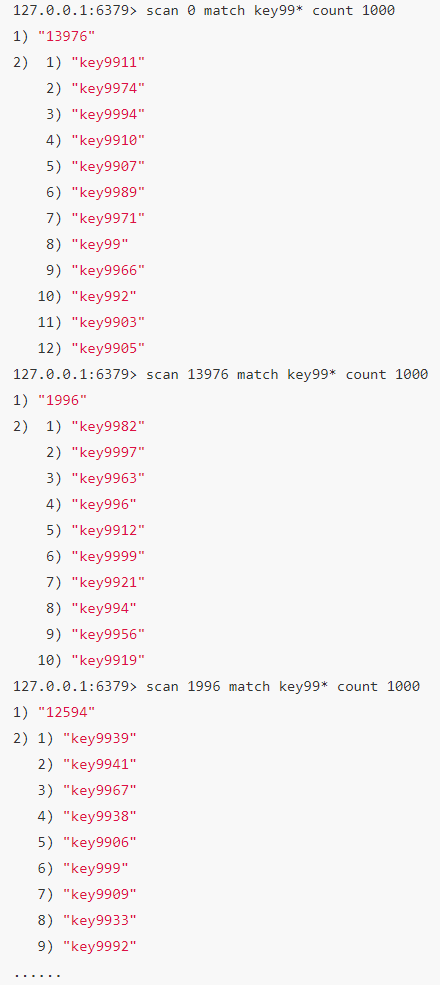


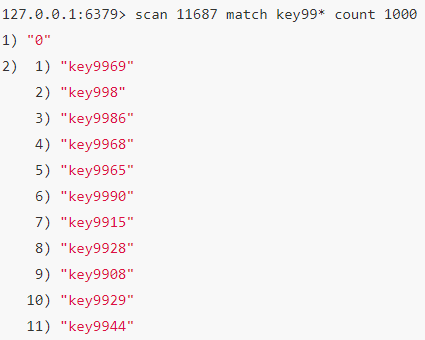
**其他高级命令**

**keys：全量遍历键**，用来列出所有满足特定正则字符串规则的key，当redis数据量比较大时，性能比较差，要避免使用

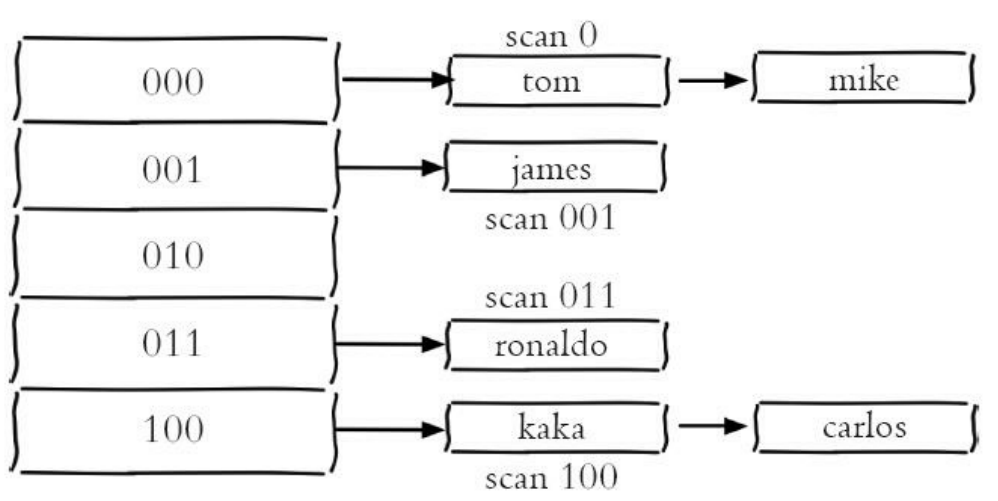


**scan：渐进式遍历键，**scan 参数提供了三个参数，第一个是 cursor 整数值，第二个是 key 的正则模式，第三个是遍历的 limit hint。第一次遍历时，cursor 值为 0，然后将返回结果中第一个整数值作为下一次遍历的 cursor。一直遍历到返回的 cursor 值为 0 时结束。





Redis存储键值对实际使用的是hashtable的数据结构



**Info：**查看redis服务运行信息，分为 9 大块，每个块都有非常多的参数，这 9 个块分别是:

Server 服务器运行的环境参数

Clients 客户端相关信息

Memory 服务器运行内存统计数据

Persistence 持久化信息

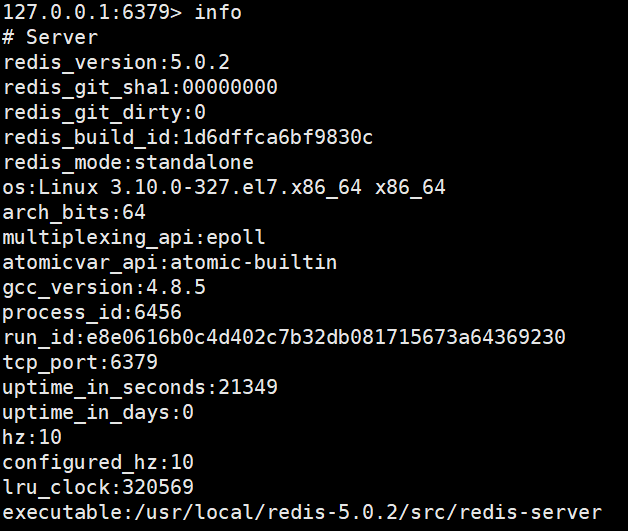
Stats 通用统计数据

Replication 主从复制相关信息

CPU CPU 使用情况

Cluster 集群信息

KeySpace 键值对统计数量信息



# 核心原理

## Redis的单线程和高性能

**Redis 单线程为什么还能这么“快”？**

因为它所有的数据都在内存中，所有的运算都是**内存级别的运算**，而且单线程避免了多线程的切换性能损耗问题。正因为 Redis 是单线程，所以要小心使用 Redis 指令，对于那些耗时的指令(比如keys)，一定要谨慎使用，一不小心就可能会导致 Redis 卡顿。

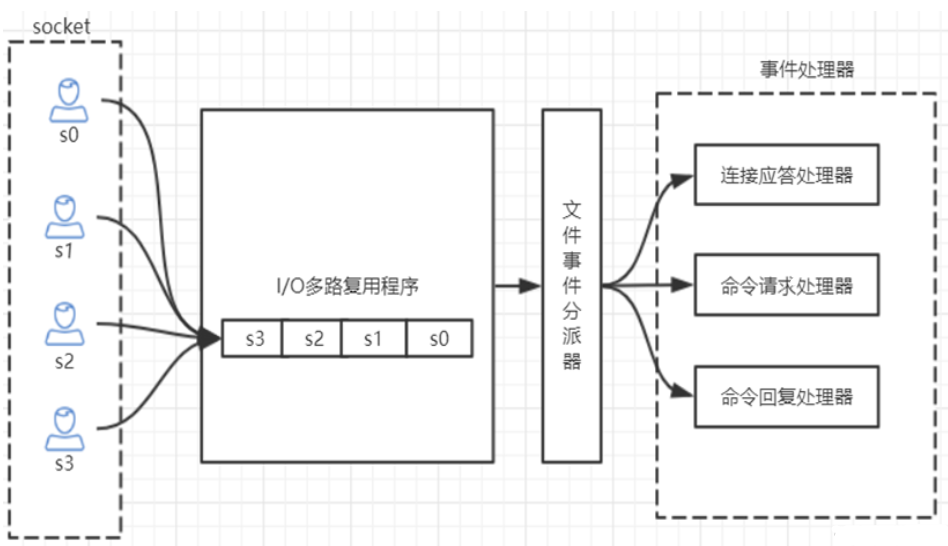
**Redis 单线程如何处理那么多的并发客户端连接？**

Redis的**IO多路复用**：redis利用**epoll**来实现IO多路复用，将连接信息和事件放到队列中，依次放到文件事件分派器，事件分派器将事件分发给事件处理器。

Nginx也是采用**IO多路复用**原理解决C10K问题

Select ,poll ,epoll,

reactor



## 

## 持久化

### RDB快照（snapshot）

在默认情况下， Redis 将内存数据库快照保存在名字为 dump.rdb 的二进制文件中。

你可以对 Redis 进行设置， 让它在“ N 秒内数据集至少有 M 个改动”这一条件被满足时， 自动保存一次数据集。

比如说， 以下设置会让 Redis 在满足“ 60 秒内有至少有 1000 个键被改动”这一条件时， 自动保存一次数据集：

# save 60 1000

### AOF（append-only file）

快照功能并不是非常耐久（durable）： 如果 Redis 因为某些原因而造成故障停机， 那么服务器将丢失最近写入、且仍未保存到快照中的那些数据。从 1.1 版本开始， Redis 增加了一种完全耐久的持久化方式： AOF 持久化，将修改的每一条指令记录进文件

你可以通过修改配置文件来打开 AOF 功能：

# appendonly yes

从现在开始， 每当 Redis 执行一个改变数据集的命令时（比如 [SET](http://redisdoc.com/string/set.html#set)）， 这个命令就会被追加到 AOF 文件的末尾。

这样的话， 当 Redis 重新启时， 程序就可以通过重新执行 AOF 文件中的命令来达到重建数据集的目的。

你可以配置 Redis 多久才将数据 fsync 到磁盘一次。

有三个选项：

* 每次有新命令追加到 AOF 文件时就执行一次 fsync ：非常慢，也非常安全。
* 每秒 fsync 一次：足够快（和使用 RDB 持久化差不多），并且在故障时只会丢失 1 秒钟的数据。
* 从不 fsync ：将数据交给操作系统来处理。更快，也更不安全的选择。

推荐（并且也是默认）的措施为每秒 fsync 一次， 这种 fsync 策略可以兼顾速度和安全性。

### RDB 和 AOF ，我应该用哪一个？

如果你非常关心你的数据， 但仍然可以承受数分钟以内的数据丢失， 那么你可以只使用 RDB 持久化。

有很多用户都只使用 AOF 持久化， 但我们并不推荐这种方式： 因为定时生成 RDB 快照（snapshot）非常便于进行数据库备份， 并且 RDB 恢复数据集的速度也要比 AOF 恢复的速度要快。

### Redis 4.0 混合持久化

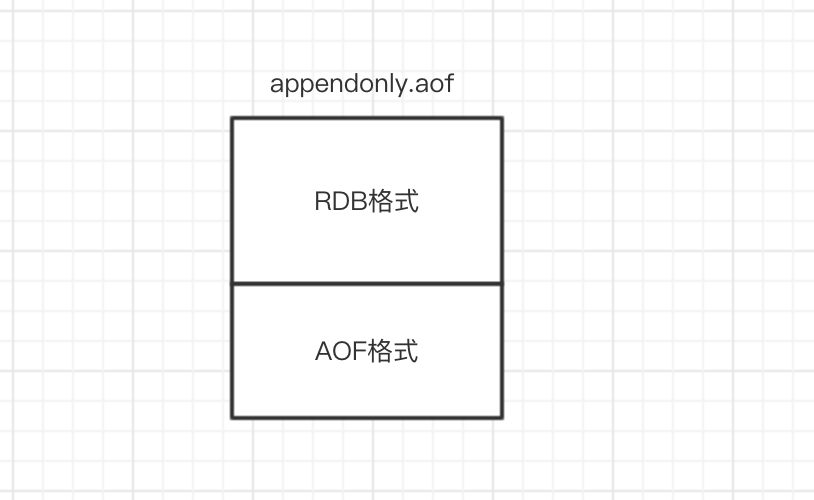
重启 Redis 时，我们很少使用 rdb 来恢复内存状态，因为会丢失大量数据。我们通常使用 AOF 日志重放，但是重放 AOF 日志性能相对 rdb 来说要慢很多，这样在 Redis 实例很大的情况下，启动需要花费很长的时间。 Redis 4.0 为了解决这个问题，带来了一个新的持久化选项——混合持久化。**AOF在重写**(aof文件里可能有太多没用指令，所以aof会定期根据内存的最新数据生成aof文件)**时将重写这一刻之前的内存rdb快照文件的内容和增量的 AOF修改内存数据的命令日志文件存在一起，都写入新的aof文件，新的文件一开始不叫appendonly.aof，等到重写完新的AOF文件才会进行改名，原子的覆盖原有的AOF文件，完成新旧两个AOF文件的替换；**

AOF根据配置规则在后台自动重写，也可以人为执行命令bgrewriteaof重写AOF。 于是在 Redis 重启的时候，可以先加载 rdb 的内容，然后再重放增量 AOF 日志就可以完全替代之前的 AOF 全量文件重放，重启效率因此大幅得到提升。

开启混合持久化：

# aof-use-rdb-preamble yes

混合持久化aof文件结构



## 缓存淘汰策略

当 Redis 内存超出物理内存限制时，内存的数据会开始和磁盘产生频繁的交换 (**swap**)。交换会让 Redis 的性能**急剧下降**，对于访问量比较频繁的 Redis 来说，这样**龟速**的存取效率基本上等于不可用。

在生产环境中我们是**不允许** Redis 出现交换行为的，为了限制最大使用内存，Redis 提供了配置参数 maxmemory 来限制内存超出期望大小。

当实际内存超出 maxmemory 时，Redis 提供了几种可选策略 (maxmemory-policy) 来让用户自己决定该如何腾出新的空间以继续提供读写服务。

**noeviction** 不会继续服务写请求 (DEL 请求可以继续服务)，读请求可以继续进行。这样可以保证不会丢失数据，但是会让线上的业务不能持续进行。这是默认的淘汰策略。

**volatile-lru** 尝试淘汰设置了过期时间的 key，最少使用的 key 优先被淘汰。没有设置过期时间的 key 不会被淘汰，这样可以保证需要持久化的数据不会突然丢失。

**volatile-ttl** 跟上面一样，除了淘汰的策略不是 LRU，而是 key 的剩余寿命 ttl 的值，ttl 越小越优先被淘汰。

**volatile-random** 跟上面一样，不过淘汰的 key 是过期 key 集合中随机的 key。

**allkeys-lru** 区别于 volatile-lru，这个策略要淘汰的 key 对象是全体的 key 集合，而不只是过期的 key 集合。这意味着没有设置过期时间的 key 也会被淘汰。

**allkeys-random** 跟上面一样，不过淘汰的策略是随机的 key。

volatile-xxx 策略只会针对带过期时间的 key 进行淘汰，allkeys-xxx 策略会对所有的 key 进行淘汰。如果你只是拿 Redis 做缓存，那应该使用 allkeys-xxx，客户端写缓存时不必携带过期时间。如果你还想同时使用 Redis 的持久化功能，那就使用 volatile-xxx 策略，这样可以保留没有设置过期时间的 key，它们是永久的 key 不会被 LRU 算法淘汰。

**noeviction:默认策略，不淘汰，如果内存已满，添加数据是报错。**

**allkeys-lru:在所有键中，选取最近最少使用的数据抛弃。**

**volatile-lru:在设置了过期时间的所有键中，选取最近最少使用的数据抛弃。**

**allkeys-random: 在所有键中，随机抛弃。**

**volatile-random: 在设置了过期时间的所有键，随机抛弃。**

**volatile-ttl:在设置了过期时间的所有键，抛弃存活时间最短的数据。**

# 数据库的一些概念

1文件系统 fat32,ntfs,ext,xf

2关系数据库：oracle,mysql,sql server,db2,统一使用SQL93标准。

3面向对象数据库，**活在理论中**。。。。。。

由于关系数据库都使用sql语言标准，所以也可以统称为：sql 数据库

但是传统的sql 数据库不能完全解决我们实际项目的所有问题，尤其是性能问题，日益突出，所以就出现了一种新的**变种**数据库：nosql数据库！！(not only sql)

数据库排名：



## Nosql数据库

代表：**redis,mongodb, HBase**

## NoSQL数据库的四大分类表格分析

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **分类** | **Examples举例** | **典型应用场景** | **数据模型** | **优点** | **缺点** |
| **键值（key-value）[3]** | Tokyo Cabinet/Tyrant, **Redis**, Voldemort, Oracle BDB | **内容缓存**，主要用于处理大量数据的高访问负载，也用于一些日志系统等等。[3] | Key 指向 Value 的键值对，通常用hash table来实现[3] | 查找速度快 | 数据无结构化，通常只被当作字符串或者二进制数据[3] |
| **列存储数据库[3]** | Cassandra, **HBase**, Riak | 分布式的文件系统 | 以列簇式存储，将同一列数据存在一起 | 查找速度快，可扩展性强，更容易进行分布式扩展 | 功能相对局限 |
| **文档型数据库[3]** | CouchDB, **MongoDb** | Web应用（与Key-Value类似，Value是结构化的，不同的是数据库能够了解Value的内容） | Key-Value对应的键值对，Value为结构化数据 | 数据结构要求不严格，表结构可变，不需要像关系型数据库一样需要预先定义表结构 | 查询性能不高，而且缺乏统一的查询语法。 |
| **图形(Graph)数据库[3]** | **Neo4J**, InfoGrid, Infinite Graph | 社交网络，推荐系统等。专注于构建关系图谱 | 图结构 | 利用图结构相关算法。比如最短路径寻址，N度关系查找等 | 很多时候需要对整个图做计算才能得出需要的信息，而且这种结构不太好做分布式的集群方案。[3] |

因此，我们总结NoSQL数据库在以下的这几种情况下比较适用：

1、数据模型比较简单；2、需要灵活性更强的IT系统；3、对数据库**性能**要求较高；4、不需要高度的数据**一致性**；5、对于给定key，比较容易映射复杂值的环境。

# 大数据(hadoop)

数据大-------》解决方法------》**分布式**！！

开源的解决方案：**hadoop**!!（了解）

jd大数据平台：

消费地位：

**女人 》 老人，小孩 》》 宠物 》》 男人**

阿里统计：

分析bar的大小：

沃尔玛分析得出：

啤酒 和 卫生巾 放一起！！