# Redis课程教案

# NoSQL数据库的发展历史简介

## 1、web系统的变迁历史

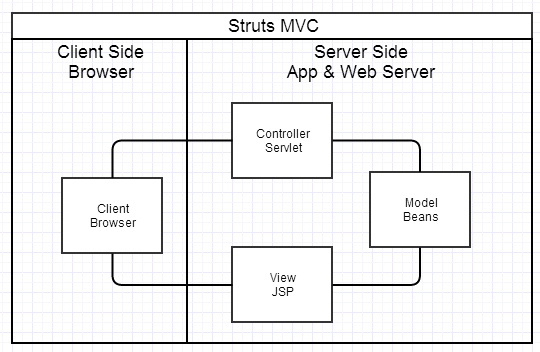
### web1.0时代简介C:\Users\admin\Desktop\2014_01_14_11.jpg

基本上就是一些简单的静态页面的渲染，不会涉及到太多的复杂业务逻辑，功能简单单一，基本上服务器性能不会有太大压力

缺点：1、Service 越来越多，调用关系变复杂，前端搭建本地环境不再是一件简单的事。考虑团队协作，往往会考虑搭建集中式的开发服务器来解决。这种解决方案对编译型的后端开发来说也许还好，但对前端开发来说并不友好。天哪，我只是想调整下按钮样式，却要本地开发、代码上传、验证生效等好几个步骤。也许习惯了也还好，但开发服务器总是不那么稳定，出问题时往往需要依赖后端开发搞定。看似仅仅是前端开发难以本地化，但这对研发效率的影响其实蛮大。

2、JSP 等代码的可维护性越来越差。JSP 非常强大，可以内嵌 Java 代码。这种强大使得前后端的职责不清晰，JSP 变成了一个灰色地带。经常为了赶项目，为了各种紧急需求，会在 JSP 里揉杂大量业务代码。积攒到一定阶段时，往往会带来大量维护成本。

### web2.0时代简介



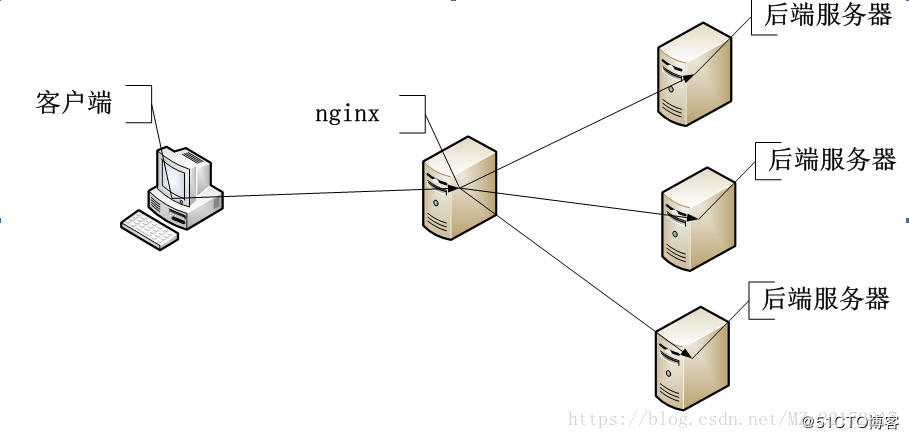
随着Web2.0的时代的到来，用户访问量大幅度提升，同时产生了大量的用户数据。加上后来的智能移动设备的普及，所有的互联网平台都面临了巨大的性能挑战。包括web服务器CPU及内存压力。数据库服务器IO压力等

为了解决服务器的性能压力问题，出现了各种各样的解决方案，最典型的就是使用MVC的架构，MVC 是个非常好的协作模式，从架构层面让开发者懂得什么代码应该写在什么地方。为了让 View 层更简单干脆，还可以选择 Velocity、Freemaker 等模板，使得模板里写不了 Java 代码。看起来是功能变弱了，但正是这种限制使得前后端分工更清晰。但是同样也会面临以下问题

1、前端开发重度依赖开发环境。这种架构下，前后端协作有两种模式：一种是前端写 demo，写好后，让后端去套模板。淘宝早期包括现在依旧有大量业务线是这种模式。好处很明显，demo 可以本地开发，很高效。不足是还需要后端套模板，有可能套错，套完后还需要前端确定，来回沟通调整的成本比较大。另一种协作模式是前端负责浏览器端的所有开发和服务器端的 View 层模板开发，支付宝是这种模式。好处是 UI 相关的代码都是前端去写就好，后端不用太关注，不足就是前端开发重度绑定后端环境，环境成为影响前端开发效率的重要因素。

2、前后端职责依旧纠缠不清。Velocity 模板还是蛮强大的，变量、逻辑、宏等特性，依旧可以通过拿到的上下文变量来实现各种业务逻辑。这样，只要前端弱势一点，往往就会被后端要求在模板层写出不少业务代码。还有一个很大的灰色地带是 Controller，页面路由等功能本应该是前端最关注的，但却是由后端来实现。Controller 本身与 Model 往往也会纠缠不清，看了让人咬牙的代码经常会出现在 Controller 层。这些问题不能全归结于程序员的素养，否则 JSP 就够了。

关于如何解决Web服务器的负载压力，其中最常用的一种方式就是使用nginx实现web集群的服务转发以及服务拆分等等



但是这样也会存在问题，后端服务器的多个tomcat之间如何解决session共享的问题，以及session存放的问题等等

为了解决session存放的问题，也有多种解决方案

方案一：存放在cookie里面。不安全，否定

方案二：存放在文件或者数据库当中。速度慢

方案三：session复制。大量session冗余，节点浪费大

方案四：使用NoSQL缓存数据库。例如redis或者memcache等，完美解决

**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
* 速度够快，能够快速的存取数据

**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（用户自定义查询条件的查询）。

**总结：用不着sql的和用了sql也不行的情况，请考虑用NoSql**

# NoSQL数据库兄弟会

## 1、memcache介绍

* 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
* 一般是作为缓存数据库辅助持久化的数据库

## 2、redis介绍

* 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
* 现在市面上用得非常多的一款内存数据库

## 3、mongoDB介绍

* 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。

## 4、列式存储HBase介绍

* + 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。

# Redis的基本介绍以及使用场景

redis官网地址：

<https://redis.io/>

中文网站

<http://www.redis.cn/>

## 3.1、redis的基本介绍

 Redis是当前比较热门的NOSQL系统之一，它是一个开源的使用ANSI c语言编写的**key-value**存储系统（区别于MySQL的二维表格的形式存储。）。和Memcache类似，但很大程度补偿了Memcache的不足。和Memcache一样，Redis数据都是缓存在计算机内存中，不同的是，Memcache只能将数据缓存到内存中，无法自动定期写入硬盘，这就表示，一断电或重启，内存清空，数据丢失。所以Memcache的应用场景适用于缓存无需持久化的数据。而Redis不同的是它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，实现数据的持久化

## 3.2、redis的适用场景

**1.取最新N个数据的操作**

比如典型的取你网站的最新文章，通过下面方式，我们可以将最新的5000条评论的ID放在Redis的List集合中，并将超出集合部分从数据库获取

* 使用LPUSH latest.comments<ID>命令，向list集合中插入数据
* 插入完成后再用LTRIM latest.comments 0 5000命令使其永远只保存最近5000个ID
* 然后我们在客户端获取某一页评论时可以用下面的逻辑（伪代码）

FUNCTION get\_latest\_comments(start,num\_items):

id\_list = redis.lrange("latest.comments",start,start+num\_items-1)

IF id\_list.length < num\_items

id\_list = SQL\_DB("SELECT ... ORDER BY time LIMIT ...")

END

RETURN id\_list

END

如果你还有不同的筛选维度，比如某个分类的最新N条，那么你可以再建一个按此分类的List，只存ID的话，Redis是非常高效的。

**2.排行榜应用，取TOP N操作**

这个需求与上面需求的不同之处在于，前面操作以时间为权重，这个是以某个条件为权重，比如按顶的次数排序，这时候就需要我们的sorted set出马了，将你要排序的值设置成sorted set的score，将具体的数据设置成相应的value，每次只需要执行一条ZADD命令即可。

**3.需要精准设定过期时间的应用**

比如你可以把上面说到的sorted set的score值设置成过期时间的时间戳，那么就可以简单地通过过期时间排序，定时清除过期数据了，不仅是清除Redis中的过期数据，你完全可以把Redis里这个过期时间当成是对数据库中数据的索引，用Redis来找出哪些数据需要过期删除，然后再精准地从数据库中删除相应的记录。

**4.计数器应用**

Redis的命令都是原子性的，你可以轻松地利用INCR，DECR命令来构建计数器系统。

**5.Uniq操作，获取某段时间所有数据排重值**

这个使用Redis的set数据结构最合适了，只需要不断地将数据往set中扔就行了，set意为集合，所以会自动排重。

**6.实时系统，反垃圾系统**

通过上面说到的set功能，你可以知道一个终端用户是否进行了某个操作，可以找到其操作的集合并进行分析统计对比等。没有做不到，只有想不到。

**7.Pub/Sub构建实时消息系统**

Redis的Pub/Sub系统可以构建实时的消息系统，比如很多用Pub/Sub构建的实时聊天系统的例子。

**8.构建队列系统**

使用list可以构建队列系统，使用sorted set甚至可以构建有优先级的队列系统。

**9.缓存**

将数据直接存放到内存中，性能优于Memcached，数据结构更多样化。

## 3.3、redis的特点

高效性：Redis读取的速度是110000次/s，写的速度是81000次/s

原子性：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。

支持多种数据结构：string（字符串）；list（列表）；hash（哈希），set（集合）；zset(有序集合)

稳定性：持久化，主从复制（集群）

其他特性：支持过期时间，支持事务，消息订阅。

# redis环境安装

## 第一步：下载redis安装包

node01服务器执行以下命令下载redis安装包

cd /export/softwares

wget http://download.redis.io/releases/redis-3.2.8.tar.gz

## 第二步：解压redis压缩包到指定目录

node01执行以下命令进行解压redis

cd /export/softwares

tar -zxvf redis-3.2.8.tar.gz -C ../servers/

## 第三步：安装C程序运行环境

node01执行以下命令安装C程序运行环境

yum -y install gcc-c++

## 第四步：安装较新版本的tcl

下载安装较新版本的tcl

### 第一种方式：使用压缩包进行安装

node01执行以下命令下载tcl安装包

cd /export/softwares

wget <http://downloads.sourceforge.net/tcl/tcl8.6.1-src.tar.gz>

解压tcl

tar -zxvf tcl8.6.1-src.tar.gz -C ../servers/

进入指定目录

cd ../servers/tcl8.6.1/unix/

./configure

make && make install

### 第二种方式，在线安装tcl

node01执行以下命令在线安装tcl

yum -y install tcl

## 第五步：进行编译redis

node01执行以下命令进行编译：

cd /export/servers/redis-3.2.8/

make MALLOC=libc  或者使用命令 make 进行编译

make test && make install

## 第六步：修改redis配置文件

node01执行以下命令修改redis配置文件

cd /export/servers/redis-3.2.8/

mkdir -p /export/servers/redis-3.2.8/logs

mkdir -p /export/servers/redis-3.2.8/redisdata

vim redis.conf

bind node01

daemonize yes

pidfile /var/run/redis\_6379.pid

logfile "/export/servers/redis-3.2.8/logs/redis.log"

dir /export/servers/redis-3.2.8/redisdata

## 第七步：启动redis

node01执行以下命令启动redis

cd /export/servers/redis-3.2.8/src

redis-server ../redis.conf

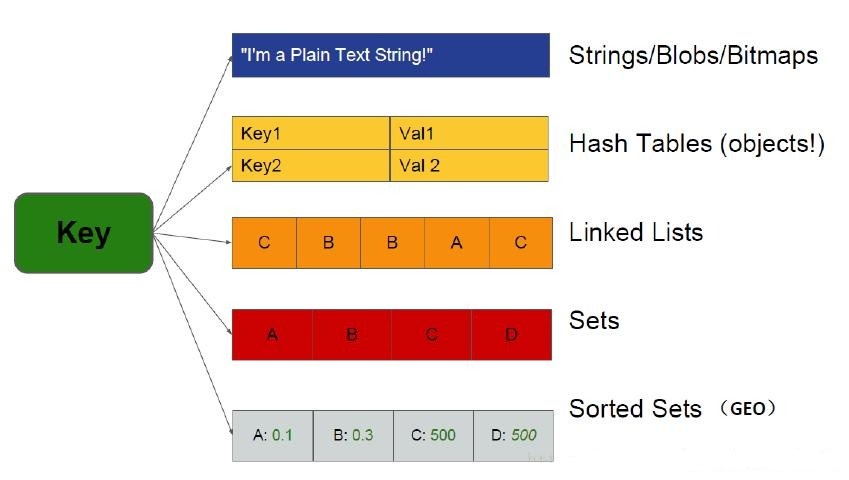
## 第八步：连接redis客户端

node01执行以下命令连接redis客户端

cd /export/servers/redis-3.2.8/src

redis-cli -h node01

# C:\Users\admin\Desktop\268981-20160226150739333-1493121959.png5、redis的数据类型

redis当中一共支持五种数据类型，分别是string字符串类型，list列表类型，集合set类型，hash表类型以及有序集合zset类型，通过这五种不同的数据类型，我们可以实现各种不同的功能，也可以应用与各种不同的场景，接下来我们来看看五种数据类型的操作语法

redis当中各种数据类型结构如上图：

redis当中各种数据类型的操作

<https://www.runoob.com/redis/redis-keys.html>

## 1、redis当中对字符串string的操作

下表列出了常用的 redis 字符串命令

|  |  |
| --- | --- |
| **序号** | **命令及描述** |
| 1 | [SET key value](https://www.runoob.com/redis/strings-set.html)  设置指定 key 的值  示例：SET hello world |
| 2 | [GET key](https://www.runoob.com/redis/strings-get.html)  获取指定 key 的值。  示例：GET hello |
| 3 | [GETRANGE key start end](https://www.runoob.com/redis/strings-getrange.html)  返回 key 中字符串值的子字符  示例：GETRANGE hello 0 3 |
| 4 | [GETSET key value](https://www.runoob.com/redis/strings-getset.html) 将给定 key 的值设为 value ，并返回 key 的旧值(old value)。  示例：GETSET hello world2 |
| 5 | [MGET key1 [key2..]](https://www.runoob.com/redis/strings-mget.html) 获取所有(一个或多个)给定 key 的值。  示例：MGET hello world |
| 6 | [SETEX key seconds value](https://www.runoob.com/redis/strings-setex.html) 将值 value 关联到 key ，并将 key 的过期时间设为 seconds (以秒为单位)。  示例：SETEX hello 10 world3 |
| 7 | [SETNX key value](https://www.runoob.com/redis/strings-setnx.html) 只有在 key 不存在时设置 key 的值。  示例：SETNX itcast redisvalue |
| 8 | [SETRANGE key offset value](https://www.runoob.com/redis/strings-setrange.html) 用 value 参数覆写给定 key 所储存的字符串值，从偏移量 offset 开始。  示例：SETRANGE itcast 0 helloredis |
| 9 | [STRLEN key](https://www.runoob.com/redis/strings-strlen.html) 返回 key 所储存的字符串值的长度。  示例：STRLEN itcast |
| 10 | [MSET key value [key value ...]](https://www.runoob.com/redis/strings-mset.html) 同时设置一个或多个 key-value 对。  示例：MSET itcast2 itcastvalue2 itcast3 itcastvalue3 |
| 12 | [MSETNX key value [key value ...]](https://www.runoob.com/redis/strings-msetnx.html)  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。  示例：MSETNX itcast4 itcastvalue4 itcast5 itcastvalue5 |
| 13 | [PSETEX key milliseconds value](https://www.runoob.com/redis/strings-psetex.html) 这个命令和 SETEX 命令相似，但它以毫秒为单位设置 key 的生存时间，而不是像 SETEX 命令那样，以秒为单位。  示例：PSETEX itcast6 6000 itcast6value |
| 14 | [INCR key](https://www.runoob.com/redis/strings-incr.html) 将 key 中储存的数字值增一。  示例：  set itcast7 1  INCR itcast7  GET itcast7 |
| 15 | [INCRBY key increment](https://www.runoob.com/redis/strings-incrby.html) 将 key 所储存的值加上给定的增量值（increment） 。  示例：INCRBY itcast7 2  get itcast7 |
| 16 | [INCRBYFLOAT key increment](https://www.runoob.com/redis/strings-incrbyfloat.html) 将 key 所储存的值加上给定的浮点增量值（increment） 。  示例：INCRBYFLOAT itcast7 0.8 |
| 17 | [DECR key](https://www.runoob.com/redis/strings-decr.html) 将 key 中储存的数字值减一。  示例：  set itcast8 1  DECR itcast8  GET itcast8 |
| 18 | [DECRBY key decrement](https://www.runoob.com/redis/strings-decrby.html) key 所储存的值减去给定的减量值（decrement） 。  示例：DECRBY itcast8 3 |
| 19 | [APPEND key value](https://www.runoob.com/redis/strings-append.html) 如果 key 已经存在并且是一个字符串， APPEND 命令将指定的 value 追加到该 key 原来值（value）的末尾。  示例：APPEND itcast8 hello |

## 2、redis当中对hash列表的操作

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。

Redis 中每个 hash 可以存储 232 - 1 键值对（40多亿）

下表列出了 redis hash 基本的相关命令：

|  |  |
| --- | --- |
| **序号** | **命令及描述** |
| 1 | [HSET key field value](https://www.runoob.com/redis/hashes-hset.html)  将哈希表 key 中的字段 field 的值设为 value 。  示例：HSET key1 field1 value1 |
| 2 | [HSETNX key field value](https://www.runoob.com/redis/hashes-hsetnx.html)  只有在字段 field 不存在时，设置哈希表字段的值。  示例：HSETNX key1 field2 value2 |
| 3 | [HMSET key field1 value1 [field2 value2 ]](https://www.runoob.com/redis/hashes-hmset.html)  同时将多个 field-value (域-值)对设置到哈希表 key 中。  示例：HMSET key1 field3 value3 field4 value4 |
| 4 | [HEXISTS key field](https://www.runoob.com/redis/hashes-hexists.html)  查看哈希表 key 中，指定的字段是否存在。  示例：  HEXISTS key1 field4  HEXISTS key1 field6 |
| 5 | [HGET key field](https://www.runoob.com/redis/hashes-hget.html)  获取存储在哈希表中指定字段的值。  示例：HGET key1 field4 |
| 6 | [HGETALL key](https://www.runoob.com/redis/hashes-hgetall.html)  获取在哈希表中指定 key 的所有字段和值  示例：HGETALL key1 |
| 7 | [HKEYS key](https://www.runoob.com/redis/hashes-hkeys.html)  获取所有哈希表中的字段  示例：HKEYS key1 |
| 8 | [HLEN key](https://www.runoob.com/redis/hashes-hlen.html)  获取哈希表中字段的数量  示例：HLEN key1 |
| 9 | [HMGET key field1 [field2]](https://www.runoob.com/redis/hashes-hmget.html)  获取所有给定字段的值  示例：HMGET key1 field1 field2 |
| 10 | [HINCRBY key field increment](https://www.runoob.com/redis/hashes-hincrby.html)  为哈希表 key 中的指定字段的整数值加上增量 increment 。  示例：  HSET key2 field1 1  HINCRBY key2 field1 1  HGET key2 field1 |
| 11 | [HINCRBYFLOAT key field increment](https://www.runoob.com/redis/hashes-hincrbyfloat.html)  为哈希表 key 中的指定字段的浮点数值加上增量 increment 。  示例：HINCRBYFLOAT key2 field1 0.8 |
| 12 | [HVALS key](https://www.runoob.com/redis/hashes-hvals.html)  获取哈希表中所有值  示例：HVALS key1 |
| 13 | [HDEL key field1 [field2]](https://www.runoob.com/redis/hashes-hdel.html)  删除一个或多个哈希表字段  示例：  HDEL key1 field1  HVALS key1 |

## 3、redis当中对list列表的操作

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）

一个列表最多可以包含 232 - 1 个元素 (4294967295, 每个列表超过40亿个元素)。

下表列出了列表相关的基本命令

|  |  |
| --- | --- |
| **序号** | **命令及描述** |
| 1 | [LPUSH key value1 [value2]](https://www.runoob.com/redis/lists-lpush.html)  将一个或多个值插入到列表头部  示例：LPUSH list1 value1 value2 |
| 2 | LRANGE key start stop  查看list当中所有的数据  示例：LRANGE list1 0 -1 |
| 3 | [LPUSHX key value](https://www.runoob.com/redis/lists-lpushx.html)  将一个值插入到已存在的列表头部  示例：LPUSHX list1 value3  LINDEX list1 0 |
| 4 | [RPUSH key value1 [value2]](https://www.runoob.com/redis/lists-rpush.html)  在列表中添加一个或多个值  示例：  RPUSH list1 value4 value5  LRANGE list1 0 -1 |
| 5 | [RPUSHX key value](https://www.runoob.com/redis/lists-rpushx.html)  为已存在的列表添加值  示例：RPUSHX list1 value6 |
| 6 | [LINSERT key BEFORE|AFTER pivot value](https://www.runoob.com/redis/lists-linsert.html)  在列表的元素前或者后插入元素  示例：LINSERT list1 BEFORE value3 beforevalue3 |
| 7 | [LINDEX key index](https://www.runoob.com/redis/lists-lindex.html)  通过索引获取列表中的元素  示例：LINDEX list1 0 |
| 8 | [LSET key index value](https://www.runoob.com/redis/lists-lset.html)  通过索引设置列表元素的值  示例：LSET list1 0 hello |
| 9 | [LLEN key](https://www.runoob.com/redis/lists-llen.html)  获取列表长度  示例：LLEN list1 |
| 10 | [LPOP key](https://www.runoob.com/redis/lists-lpop.html)  移出并获取列表的第一个元素  示例：LPOP list1 |
| 11 | [RPOP key](https://www.runoob.com/redis/lists-rpop.html)  移除列表的最后一个元素，返回值为移除的元素。  示例：RPOP list1 |
| 12 | [BLPOP key1 [key2 ] timeout](https://www.runoob.com/redis/lists-blpop.html)  移出并获取列表的第一个元素， 如果列表没有元素会阻塞列表直到等待超时或发现可弹出元素为止。  示例：BLPOP list1 2000 |
| 13 | [BRPOP key1 [key2 ] timeout](https://www.runoob.com/redis/lists-brpop.html)  移出并获取列表的最后一个元素， 如果列表没有元素会阻塞列表直到等待超时或发现可弹出元素为止。  示例：BRPOP list1 2000 |
| 14 | [RPOPLPUSH source destination](https://www.runoob.com/redis/lists-rpoplpush.html)  移除列表的最后一个元素，并将该元素添加到另一个列表并返回  示例：RPOPLPUSH list1 list2 |
| 15 | [BRPOPLPUSH source destination timeout](https://www.runoob.com/redis/lists-brpoplpush.html)  从列表中弹出一个值，将弹出的元素插入到另外一个列表中并返回它； 如果列表没有元素会阻塞列表直到等待超时或发现可弹出元素为止。  示例：BRPOPLPUSH list1 list2 2000 |
| 16 | [LTRIM key start stop](https://www.runoob.com/redis/lists-ltrim.html)  对一个列表进行修剪(trim)，就是说，让列表只保留指定区间内的元素，不在指定区间之内的元素都将被删除。  示例：LTRIM list1 0 2 |
| 17 | DEL key1 key2  删除指定key的列表  示例：DEL list2 |

## 4、redis操作set集合

redis 的 Set 是 String 类型的无序集合。集合成员是唯一的，这就意味着集合中不能出现重复的数据。

Redis 中集合是通过哈希表实现的，所以添加，删除，查找的复杂度都是 O(1)。

集合中最大的成员数为 232 - 1 (4294967295, 每个集合可存储40多亿个成员)。

下表列出了 Redis 集合基本命令：

|  |  |
| --- | --- |
| **序号** | **命令及描述** |
| 1 | [SADD key member1 [member2]](https://www.runoob.com/redis/sets-sadd.html)  向集合添加一个或多个成员  示例：SADD set1 setvalue1 setvalue2 |
| 2 | [SMEMBERS key](https://www.runoob.com/redis/sets-smembers.html)  返回集合中的所有成员  示例：SMEMBERS set1 |
| 3 | [SCARD key](https://www.runoob.com/redis/sets-scard.html)  获取集合的成员数  示例：SCARD set1 |
| 4 | [SDIFF key1 [key2]](https://www.runoob.com/redis/sets-sdiff.html)  返回给定所有集合的差集  示例：  SADD set2 setvalue2 setvalue3  SDIFF set1 set2 |
| 5 | [SDIFFSTORE destination key1 [key2]](https://www.runoob.com/redis/sets-sdiffstore.html)  返回给定所有集合的差集并存储在 destination 中  示例：SDIFFSTORE set3 set1 set2 |
| 6 | [SINTER key1 [key2]](https://www.runoob.com/redis/sets-sinter.html)  返回给定所有集合的交集  示例：SINTER set1 set2 |
| 7 | [SINTERSTORE destination key1 [key2]](https://www.runoob.com/redis/sets-sinterstore.html)  返回给定所有集合的交集并存储在 destination 中  示例：SINTERSTORE set4 set1 set2 |
| 8 | [SISMEMBER key member](https://www.runoob.com/redis/sets-sismember.html)  判断 member 元素是否是集合 key 的成员  示例：SISMEMBER set1 setvalue1 |
| 9 | [SMOVE source destination member](https://www.runoob.com/redis/sets-smove.html)  将 member 元素从 source 集合移动到 destination 集合  示例：SMOVE set1 set2 setvalue1 |
| 10 | [SPOP key](https://www.runoob.com/redis/sets-spop.html)  移除并返回集合中的一个随机元素  示例：SPOP set2 |
| 11 | [SRANDMEMBER key [count]](https://www.runoob.com/redis/sets-srandmember.html)  返回集合中一个或多个随机数  示例：SRANDMEMBER set2 2 |
| 12 | [SREM key member1 [member2]](https://www.runoob.com/redis/sets-srem.html)  移除集合中一个或多个成员  示例：SREM set2 setvalue1 |
| 13 | [SUNION key1 [key2]](https://www.runoob.com/redis/sets-sunion.html)  返回所有给定集合的并集  示例：SUNION set1 set2 |
| 14 | [SUNIONSTORE destination key1 [key2]](https://www.runoob.com/redis/sets-sunionstore.html)  所有给定集合的并集存储在 destination 集合中  示例：SUNIONSTORE set5 set1 set2 |

## 5、redis中对key的操作

下表给出了与 Redis 键相关的基本命令：

|  |  |
| --- | --- |
| **序号** | **命令及描述** |
| 1 | [DEL key](https://www.runoob.com/redis/keys-del.html) 该命令用于在 key 存在时删除 key。  示例：del itcast5 |
| 2 | [DUMP key](https://www.runoob.com/redis/keys-dump.html)  序列化给定 key ，并返回被序列化的值。  示例：DUMP key1 |
| 3 | [EXISTS key](https://www.runoob.com/redis/keys-exists.html)  检查给定 key 是否存在。  示例：exists itcast |
| 4 | [EXPIRE key](https://www.runoob.com/redis/keys-expire.html) seconds 为给定 key 设置过期时间，以秒计。  示例：expire itcast 5 |
| 6 | [PEXPIRE key milliseconds](https://www.runoob.com/redis/keys-pexpire.html)  设置 key 的过期时间以毫秒计。  示例：PEXPIRE set2 3000000 |
| 8 | [KEYS pattern](https://www.runoob.com/redis/keys-keys.html)  查找所有符合给定模式( pattern)的 key 。  示例：keys \* |
| 10 | [PERSIST key](https://www.runoob.com/redis/keys-persist.html)  移除 key 的过期时间，key 将持久保持。  示例：persist set2 |
| 11 | [PTTL key](https://www.runoob.com/redis/keys-pttl.html)  以毫秒为单位返回 key 的剩余的过期时间。  示例：pttl set2 |
| 12 | [TTL key](https://www.runoob.com/redis/keys-ttl.html)  以秒为单位，返回给定 key 的剩余生存时间(TTL, time to live)。  示例：ttl set2 |
| 13 | [RANDOMKEY](https://www.runoob.com/redis/keys-randomkey.html)  从当前数据库中随机返回一个 key 。  示例： randomkey |
| 14 | [RENAME key newkey](https://www.runoob.com/redis/keys-rename.html)  修改 key 的名称  示例：rename set5 set8 |
| 15 | [RENAMENX key newkey](https://www.runoob.com/redis/keys-renamenx.html)  仅当 newkey 不存在时，将 key 改名为 newkey 。  示例：renamenx set8 set10 |
| 16 | [TYPE key](https://www.runoob.com/redis/keys-type.html)  返回 key 所储存的值的类型。  示例：type set10 |

# 6、redis的javaAPI操作

redis不仅可以通过命令行进行操作，同时redis也可以通过javaAPI进行操作，我们可以通过使用javaAPI来对redis数据库当中的各种数据类型进行操作

## 第一步：创建maven工程并导入jar包

<dependencies>

<dependency>

<groupId>redis.clients</groupId>

<artifactId>jedis</artifactId>

<version>2.9.0</version>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.testng</groupId>

<artifactId>testng</artifactId>

<version>6.14.3</version>

<scope>test</scope>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.0</version>

<configuration>

<source>1.8</source>

<target>1.8</target>

<encoding>UTF-8</encoding>

<!-- <verbal>true</verbal>-->

</configuration>

</plugin>

</plugins>

</build>

## 第二步：连接以及关闭redis客户端

**private** JedisPool **jedisPool**;  
**private** JedisPoolConfig **config**;  
  
@BeforeTest  
**public void** redisConnectionPool(){  
 **config** = **new** JedisPoolConfig();  
 **config**.setMaxIdle(10);  
 **config**.setMaxWaitMillis(3000);  
 **config**.setMaxTotal(50);  
 **config**.setMinIdle(5);  
 **jedisPool** = **new** JedisPool(**config**, **"node01"**, 6379);  
}  
@AfterTest  
**public void** closePool(){  
 **jedisPool**.close();  
}

## 第三步：redis的javaAPI操作

### 操作string类型数据

*/\*\*  
 \* 添加string类型数据  
 \*/*@Test  
**public void** addStr(){  
 Jedis resource = **jedisPool**.getResource();  
 *//添加* resource.set(**"jediskey"**, **"jedisvalue"**);  
 *//查询* String jediskey = resource.get(**"jediskey"**);  
 System.***out***.println(jediskey);  
 *//修改* resource.set(**"jediskey"**,**"jedisvalueUpdate"**);  
 *//删除* resource.del(**"jediskey"**);  
 *//实现整型数据的增长操作* resource.incr(**"jincr"**);  
 resource.incrBy(**"jincr"**,3);  
 String jincr = resource.get(**"jincr"**);  
 System.***out***.println(jincr);  
 resource.close();  
}

### 操作hash列表类型数据

*/\*\*  
 \* 操作hash类型数据  
 \*/*@Test  
**public void** hashOperate(){  
 Jedis resource = **jedisPool**.getResource();  
 *//添加数据* resource.hset(**"jhsetkey"**,**"jmapkey"**,**"jmapvalue"**);  
 resource.hset(**"jhsetkey"**,**"jmapkey2"**,**"jmapvalue2"**);  
 *//获取所有数据* Map<String, String> jhsetkey = resource.hgetAll(**"jhsetkey"**);  
 **for** (String s : jhsetkey.keySet()) {  
 System.***out***.println(s);  
 }  
 *//修改数据* resource.hset(**"jhsetkey"**,**"jmapkey2"**,**"jmapvalueupdate2"**);  
 Map<String, String> jhsetkey2 = resource.hgetAll(**"jhsetkey"**);  
 **for** (String s : jhsetkey2.keySet()) {  
 System.***out***.println(**"修改数据打印"**+s);  
 }  
 *//删除数据* resource.del(**"jhsetkey"**);  
  
 Set<String> jhsetkey1 = resource.keys(**"jhsetkey"**);  
 **for** (String result : jhsetkey1) {  
 System.***out***.println(result);  
 }  
}

### 操作list类型数据

*/\*\*  
 \* 操作list类型的数据  
 \*/*@Test  
**public void** listOperate(){  
 Jedis resource = **jedisPool**.getResource();  
 *//从左边插入元素* resource.lpush(**"listkey"**,**"listvalue1"**,**"listvalue1"**,**"listvalue2"**);  
  
 *//从右边移除元素* resource.rpop(**"listkey"**);  
 *//获取所有值* List<String> listkey = resource.lrange(**"listkey"**, 0, -1);  
 **for** (String s : listkey) {  
 System.***out***.println(s);  
 }  
 resource.close();  
}

### 操作set类型的数据

*/\*\*  
 \* set类型数据操作  
 \*/*@Test  
**public void** setOperate(){  
 Jedis resource = **jedisPool**.getResource();  
 *//添加数据* resource.sadd(**"setkey"**, **"setvalue1"**, **"setvalue1"**, **"setvalue2"**, **"setvalue3"**);  
 *//查询数据* Set<String> setkey = resource.smembers(**"setkey"**);  
 **for** (String s : setkey) {  
 System.***out***.println(s);  
 }  
 *//移除掉一个数据* resource.srem(**"setkey"**,**"setvalue3"**);  
 resource.close();  
}

# 7、redis的持久化

由于redis是一个内存数据库，所有的数据都是保存在内存当中的，内存当中的数据极易丢失，所以redis的数据持久化就显得尤为重要，在redis当中，提供了两种数据持久化的方式，分别为RDB以及AOF，且redis默认开启的数据持久化方式为RDB方式，接下来我们就分别来看下两种方式的配置吧

## 1、RDB持久化方案介绍

### RDB方案介绍

Redis会定期保存数据快照至一个rbd文件中，并在启动时自动加载rdb文件，恢复之前保存的数据。可以在配置文件中配置Redis进行快照保存的时机：

save [seconds] [changes]

意为在[seconds]秒内如果发生了[changes]次数据修改，则进行一次RDB快照保存，例如

save 60 100

会让Redis每60秒检查一次数据变更情况，如果发生了100次或以上的数据变更，则进行RDB快照保存。可以配置多条save指令，让Redis执行多级的快照保存策略。Redis默认开启RDB快照。也可以通过SAVE或者BGSAVE命令手动触发RDB快照保存。

SAVE 和 BGSAVE 两个命令都会调用 rdbSave 函数，但它们调用的方式各有不同：

* SAVE 直接调用 rdbSave ，阻塞 Redis 主进程，直到保存完成为止。在主进程阻塞期间，服务器不能处理客户端的任何请求。
* BGSAVE 则 fork 出一个子进程，子进程负责调用 rdbSave ，并在保存完成之后向主进程发送信号，通知保存已完成。 Redis 服务器在BGSAVE 执行期间仍然可以继续处理客户端的请求。

### RDB方案优点

1. 对性能影响最小。如前文所述，Redis在保存RDB快照时会fork出子进程进行，几乎不影响Redis处理客户端请求的效率。
2. 每次快照会生成一个完整的数据快照文件，所以可以辅以其他手段保存多个时间点的快照（例如把每天0点的快照备份至其他存储媒介中），作为非常可靠的灾难恢复手段。
3. 使用RDB文件进行数据恢复比使用AOF要快很多

### RDB方案缺点

1. 快照是定期生成的，所以在Redis crash时或多或少会丢失一部分数据。
2. 如果数据集非常大且CPU不够强（比如单核CPU），Redis在fork子进程时可能会消耗相对较长的时间，影响Redis对外提供服务的能力。

### RDB方案配置

修改redis的配置文件

cd /export/servers/redis-3.2.8/

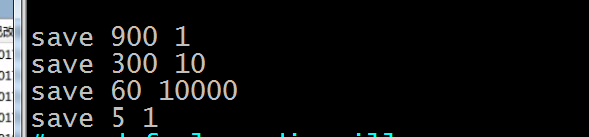
vim redis.conf

save 900 1

save 300 10

save 60 10000

save 5 1



重新启动redis服务

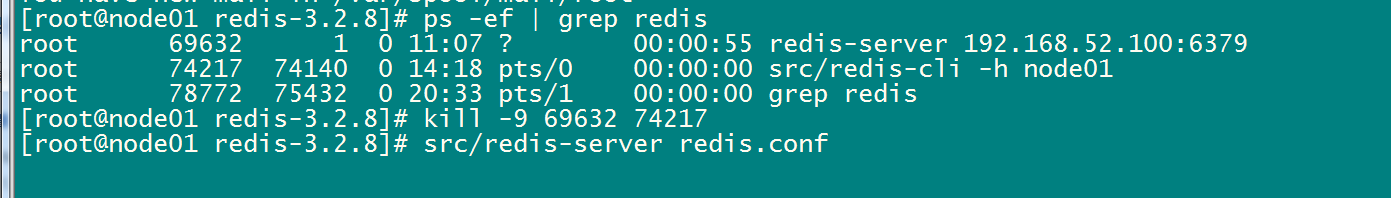
每次生成新的dump.rdb都会覆盖掉之前的老的快照

ps -ef | grep redis

kill -9 69632 74217

src/redis-server redis.conf

root 69632 1 0 11:07 ? 00:00:55 redis-server 192.168.52.100:6379



## 2、AOF持久化方案介绍

### AOF方案介绍：

采用AOF持久方式时，Redis会把每一个写请求都记录在一个日志文件里。在Redis重启时，会把AOF文件中记录的所有写操作顺序执行一遍，确保数据恢复到最新。AOF默认是关闭的，如要开启，进行如下配置：

appendonly yes

AOF提供了三种fsync配置，always/everysec/no，通过配置项[appendfsync]指定：

appendfsync no：不进行fsync，将flush文件的时机交给OS决定，速度最快appendfsync always：每写入一条日志就进行一次fsync操作，数据安全性最高，但速度最慢

appendfsync everysec：折中的做法，交由后台线程每秒fsync一次

随着AOF不断地记录写操作日志，因为所有的操作都会记录，所以必定会出现一些无用的日志。大量无用的日志会让AOF文件过大，也会让数据恢复的时间过长。不过Redis提供了AOF rewrite功能，可以重写AOF文件，只保留能够把数据恢复到最新状态的最小写操作集。

AOF rewrite可以通过BGREWRITEAOF命令触发，也可以配置Redis定期自动进行：

auto-aof-rewrite-percentage 100auto-aof-rewrite-min-size 64mb

上面两行配置的含义是，Redis在每次AOF rewrite时，会记录完成rewrite后的AOF日志大小，当AOF日志大小在该基础上增长了100%后，自动进行AOF rewrite。同时如果增长的大小没有达到64mb，则不会进行rewrite。

### AOF优点：

1. 最安全，在启用appendfsync always时，任何已写入的数据都不会丢失，使用在启用appendfsync everysec也至多只会丢失1秒的数据
2. AOF文件在发生断电等问题时也不会损坏，即使出现了某条日志只写入了一半的情况，也可以使用redis-check-aof工具轻松修复。
3. AOF文件易读，可修改，在进行了某些错误的数据清除操作后，只要AOF文件没有rewrite，就可以把AOF文件备份出来，把错误的命令删除，然后恢复数据。

**AOF的缺点：**

1. AOF文件通常比RDB文件更大
2. 性能消耗比RDB高
3. 数据恢复速度比RDB慢

Redis的数据持久化工作本身就会带来延迟，需要根据数据的安全级别和性能要求制定合理的持久化策略：

AOF + fsync always的设置虽然能够绝对确保数据安全，但每个操作都会触发一次fsync，会对Redis的性能有比较明显的影响

AOF + fsync every second是比较好的折中方案，每秒fsync一次

AOF + fsync never会提供AOF持久化方案下的最优性能使用RDB持久化通常会提供比使用AOF更高的性能，但需要注意RDB的策略配置

每一次RDB快照和AOF Rewrite都需要Redis主进程进行fork操作。fork操作本身可能会产生较高的耗时，与CPU和Redis占用的内存大小有关。根据具体的情况合理配置RDB快照和AOF Rewrite时机，避免过于频繁的fork带来的延迟

Redis在fork子进程时需要将内存分页表拷贝至子进程，以占用了24GB内存的Redis实例为例，共需要拷贝24GB / 4kB \* 8 = 48MB的数据。在使用单Xeon 2.27Ghz的物理机上，这一fork操作耗时216ms。

### AOF方案配置

在redis中，aof的持久化机制默认是关闭的

AOF持久化，默认是关闭的，默认是打开RDB持久化

appendonly yes，可以打开AOF持久化机制，在生产环境里面，一般来说AOF都是要打开的，除非你说随便丢个几分钟的数据也无所谓

打开AOF持久化机制之后，redis每次接收到一条写命令，就会写入日志文件中，当然是先写入os cache的，然后每隔一定时间再fsync一下

而且即使AOF和RDB都开启了，redis重启的时候，也是优先通过AOF进行数据恢复的，因为aof数据比较完整

可以配置AOF的fsync策略，有三种策略可以选择，一种是每次写入一条数据就执行一次fsync; 一种是每隔一秒执行一次fsync; 一种是不主动执行fsync

always: 每次写入一条数据，立即将这个数据对应的写日志fsync到磁盘上去，性能非常非常差，吞吐量很低; 确保说redis里的数据一条都不丢，那就只能这样了

在redis当中默认的AOF持久化机制都是关闭的

#### 配置redis的AOF持久化机制方式

cd /export/servers/redis-3.2.8

vim redis.conf

appendonly yes

# appendfsync always

appendfsync everysec

# appendfsync no

# 8、redis的主从复制架构

在Redis中，用户可以通过执行SLAVEOF命令或者设置slaveof选项，让一个服务器去复制（replicate）另一个服务器，我们称呼被复制的服务器为主服务器（master），而对主服务器进行复制的服务器则被称为从服务器（slave），如图所示。

C:\Users\admin\Desktop\071722389872883.png

使用主从复制这种模式，实现node01作为主节点，node02与node03作为从节点，并且将node01所有的数据全部都同步到node02与node03服务器

## 第一步：node02与node03服务器安装redis

### node02与node03执行以下命令安装依赖环境

yum -y install gcc-c++

### node02与node03上传redis压缩包

在node02与node03服务器上面上传redis压缩包，然后进行解压

将安装包上传到/export/softwares路径下

cd /export/softwares

tar -zxvf redis-3.2.8.tar.gz -C ../servers/

### node02与node03服务器安装tcl

node02与node03服务器执行以下命令在线装TCL

yum -y install tcl

### node02与node03服务器编译redis

node02与node03执行以下命令进行编译redis

执行以下命令进行编译：

cd /export/servers/redis-3.2.8/

make MALLOC=libc  或者使用命令 make 进行编译

make test && make install

### node02与node03服务器修改redis配置文件

#### node02服务器修改配置文件命令如下

执行以下命令修改redis配置文件

cd /export/servers/redis-3.2.8/

mkdir -p /export/servers/redis-3.2.8/logs

mkdir -p /export/servers/redis-3.2.8/redisdata

vim redis.conf

bind node02

daemonize yes

pidfile /var/run/redis\_6379.pid

logfile "/export/servers/redis-3.2.8/logs/redis.log"

dir /export/servers/redis-3.2.8/redisdata

slaveof node01 6379

#### node03服务器修改配置文件命令如下

执行以下命令修改redis配置文件

cd /export/servers/redis-3.2.8/

mkdir -p /export/servers/redis-3.2.8/logs

mkdir -p /export/servers/redis-3.2.8/redisdata

vim redis.conf

bind node03

daemonize yes

pidfile /var/run/redis\_6379.pid

logfile "/export/servers/redis-3.2.8/logs/redis.log"

dir /export/servers/redis-3.2.8/redisdata

slaveof node01 6379

## 第二步、启动node02与node03机器的redis服务

node02执行以下命令启动redis服务

cd /export/servers/redis-3.2.8/src

redis-server ../redis.conf

node03执行以下命令启动redis服务

cd /export/servers/redis-3.2.8/src

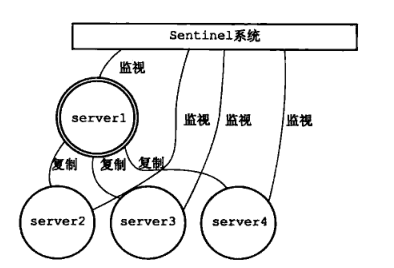
redis-server ../redis.conf

启动成功便可以实现redis的主从复制，node01可以读写操作，node02与node03只支持读取操作。

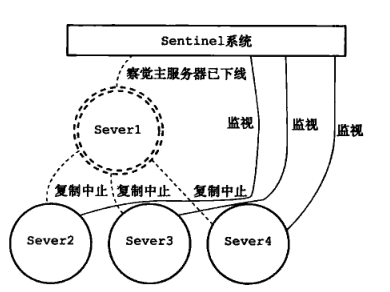
# 9、redis当中的Sentinel架构

　　Sentinel（哨兵）是Redis 的高可用性解决方案：由一个或多个Sentinel 实例 组成的Sentinel 系统可以监视任意多个主服务器，以及这些主服务器属下的所有从服务器，并在被监视的主服务器进入下线状态时，自动将下线主服务器属下的某个从服务器升级为新的主服务器。

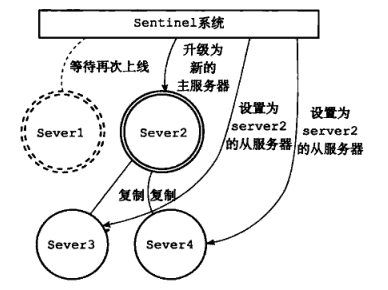
例如：



在Server1 掉线后：



升级Server2 为新的主服务器：



## 第一步：三台机器修改哨兵配置文件

三台机器执行以下命令修改redis的哨兵配置文件

cd /export/servers/redis-3.2.8

vim sentinel.conf

# 配置监听的主服务器，这里sentinel monitor代表监控，mymaster代表服务器的名称，可以自定义，192.168.11.128代表监控的主服务器，6379代表端口，2代表只有两个或两个以上的哨兵认为主服务器不可用的时候，才会进行failover操作。

#修改bind配置，每台机器修改为自己对应的主机名

bind node01

#配置sentinel服务后台运行

daemonize yes

#修改三台机器监控的主节点，现在主节点是node01服务器

sentinel monitor mymaster node01 6379 2

# sentinel author-pass定义服务的密码，mymaster是服务名称，123456是Redis服务器密码

# sentinel auth-pass <master-name> <password>

## 第二步：三台机器启动哨兵服务

cd /export/servers/redis-3.2.8

src/redis-sentinel sentinel.conf

## 第三步：node01服务器杀死redis服务进程

使用kill -9命令杀死redis服务进程，模拟redis故障宕机情况

过一段时间之后，就会在node02与node03服务器选择一台服务器来切换为主节点

## 第四步：redis的sentinel模式代码开发连接

*/\*\*  
 \* 测试Redis哨兵模式  
 \** ***@author*** *liu  
 \*/* @Test  
 **public void** testSentinel() {  
 JedisPoolConfig jedisPoolConfig = **new** JedisPoolConfig();  
 jedisPoolConfig.setMaxTotal(10);  
 jedisPoolConfig.setMaxIdle(5);  
 jedisPoolConfig.setMinIdle(5);  
 *// 哨兵信息* Set<String> sentinels = **new** HashSet<>(Arrays.*asList*(**"node01:26379"**,  
 **"node02:26379"**,**"node03:26379"**));  
 *// 创建连接池* JedisSentinelPool pool = **new** JedisSentinelPool(**"mymaster"**, sentinels,jedisPoolConfig);  
 *// 获取客户端* Jedis jedis = pool.getResource();  
 *// 执行两个命令* jedis.set(**"mykey"**, **"myvalue"**);  
 String value = jedis.get(**"mykey"**);  
 System.***out***.println(value);  
 }

# 10、redis集群

## 1、redis集群的介绍

Redis 集群是一个提供在多个Redis节点之间共享数据的程序集。

Redis 集群并不支持同时处理多个键的 Redis 命令，因为这需要在多个节点间移动数据，这样会降低redis集群的性能，在高负载的情况下可能会导致不可预料的错误。

Redis 集群通过分区来提供一定程度的可用性，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。

Redis 集群的优势:

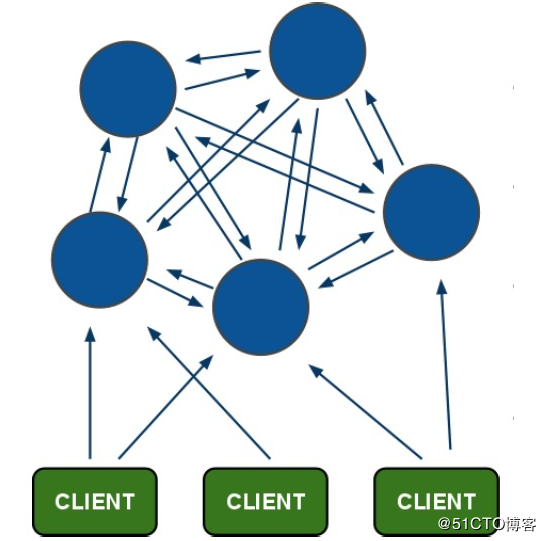
1.缓存永不宕机：启动集群，永远让集群的一部分起作用。主节点失效了子节点能迅速改变角色成为主节点，整个集群的部分节点失败或者不可达的情况下能够继续处理命令；

2.迅速恢复数据：持久化数据，能在宕机后迅速解决数据丢失的问题；

3.Redis可以使用所有机器的内存，变相扩展性能；

4.使Redis的计算能力通过简单地增加服务器得到成倍提升,Redis的网络带宽也会随着计算机和网卡的增加而成倍增长；

5.Redis集群没有中心节点，不会因为某个节点成为整个集群的性能瓶颈;

6.异步处理数据，实现快速读写。

## 2、redis集群环境搭建

由于redis集群当中最少需要三个主节点，每个主节点，最少需要一个对应的从节点，所以搭建redis集群最少需要三主三从的配置，所以redis集群最少需要6台redis的实例，我们这里使用三台机器，每台服务器上面运行两个redis的实例。我们这里使用node01服务器，通过配置不同的端口，实现redis集群的环境搭建

### 第一步：node01服务器解压redis压缩包

node01执行以下命令重新解压redis压缩包到/export路径下

cd /export/softwares/

tar -zxf redis-3.2.8.tar.gz -C /export/

### 第二步：安装redis必须依赖环境并进行编译

node01执行以下命令安装redis的依赖环境

yum -y install gcc-c++ tcl

对redis进行编译

cd /export/redis-3.2.8

make && make install

### 第三步：创建redis不同实例的配置文件夹

创建文件夹，并将redis的配置文件拷贝到以下这些目录

cd /export/redis-3.2.8

mkdir -p /export/redis-3.2.8/clusters/7001

mkdir -p /export/redis-3.2.8/clusters/7002

mkdir -p /export/redis-3.2.8/clusters/7003

mkdir -p /export/redis-3.2.8/clusters/7004

mkdir -p /export/redis-3.2.8/clusters/7005

mkdir -p /export/redis-3.2.8/clusters/7006

### 第四步：修改redis的六个配置文件

node01执行以下命令修改redis的配置文件

mkdir -p /export/redis-3.2.8/logs

mkdir -p /export/redis-3.2.8/redisdata/7001

mkdir -p /export/redis-3.2.8/redisdata/7002

mkdir -p /export/redis-3.2.8/redisdata/7003

mkdir -p /export/redis-3.2.8/redisdata/7004

mkdir -p /export/redis-3.2.8/redisdata/7005

mkdir -p /export/redis-3.2.8/redisdata/7006

第一个配置文件修改

vim /export/redis-3.2.8/redis.conf

bind node01

port 7001

cluster-enabled yes

cluster-config-file nodes-7001.conf

cluster-node-timeout 5000

appendonly yes

daemonize yes

pidfile /var/run/redis\_7001.pid

logfile "/export/redis-3.2.8/logs/7001.log"

dir /export/redis-3.2.8/redisdata/7001

将修改后的文件拷贝到对应的文件夹下面去

cp /export/redis-3.2.8/redis.conf /export/redis-3.2.8/clusters/7001

第二个配置文件修改

cp /export/redis-3.2.8/redis.conf /export/redis-3.2.8/clusters/7002

vim /export/redis-3.2.8/clusters/7002/redis.conf

bind node01

port 7002

cluster-enabled yes

cluster-config-file nodes-7002.conf

cluster-node-timeout 5000

appendonly yes

daemonize yes

pidfile /var/run/redis\_7002.pid

logfile "/export/redis-3.2.8/logs/7002.log"

dir /export/redis-3.2.8/redisdata/7002

第三个配置文件修改

cp /export/redis-3.2.8/redis.conf /export/redis-3.2.8/clusters/7003

vim /export/redis-3.2.8/clusters/7003/redis.conf

bind node01

port 7003

cluster-enabled yes

cluster-config-file nodes-7003.conf

cluster-node-timeout 5000

appendonly yes

daemonize yes

pidfile /var/run/redis\_7003.pid

logfile "/export/redis-3.2.8/logs/7003.log"

dir /export/redis-3.2.8/redisdata/7003

第四个配置文件修改

cp /export/redis-3.2.8/redis.conf /export/redis-3.2.8/clusters/7004

vim /export/redis-3.2.8/clusters/7004/redis.conf

bind node01

port 7004

cluster-enabled yes

cluster-config-file nodes-7004.conf

cluster-node-timeout 5000

appendonly yes

daemonize yes

pidfile /var/run/redis\_7004.pid

logfile "/export/redis-3.2.8/logs/7004.log"

dir /export/servers/redis-3.2.8/redisdata/7004

第五个配置文件修改

cp /export/redis-3.2.8/redis.conf /export/redis-3.2.8/clusters/7005

vim /export/redis-3.2.8/clusters/7005/redis.conf

bind node01

port 7005

cluster-enabled yes

cluster-config-file nodes-7005.conf

cluster-node-timeout 5000

appendonly yes

daemonize yes

pidfile /var/run/redis\_7005.pid

logfile "/export/redis-3.2.8/logs/7005.log"

dir /export/servers/redis-3.2.8/redisdata/7005

第六个配置文件修改

cp /export/redis-3.2.8/redis.conf /export/redis-3.2.8/clusters/7006

vim /export/redis-3.2.8/clusters/7006/redis.conf

bind node01

port 7006

cluster-enabled yes

cluster-config-file nodes-7006.conf

cluster-node-timeout 5000

appendonly yes

daemonize yes

pidfile /var/run/redis\_7006.pid

logfile "/export/redis-3.2.8/logs/7006.log"

dir /export/servers/redis-3.2.8/redisdata/7006

### 第五步：启动redis进程

node01执行以下命令来启动redis集群

cd /export/redis-3.2.8

src/redis-server clusters/7001/redis.conf

src/redis-server clusters/7002/redis.conf

src/redis-server clusters/7003/redis.conf

src/redis-server clusters/7004/redis.conf

src/redis-server clusters/7005/redis.conf

src/redis-server clusters/7006/redis.conf

### 第六步：安装ruby运行环境

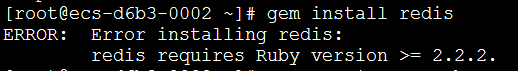
node01执行以下命令，安装ruby运行环境，因为redis集群的启动需要借助ruby的环境

yum install ruby

yum install rubygems

gem install redis

这个时候可能会报错，如下：



这时，需要升级Ruby版本

node01执行以下命令升级ruby版本

cd /export/redis-3.2.8

gpg --keyserver hkp://keys.gnupg.net --recv-keys 409B6B1796C275462A1703113804BB82D39DC0E3 7D2BAF1CF37B13E2069D6956105BD0E739499BDB

curl -sSL https://get.rvm.io | bash -s stable

source /etc/profile.d/rvm.sh

rvm list known

rvm install 2.4.1

### 第七步：创建redis集群

node01执行以下命令创建redis集群

cd /export/redis-3.2.8

gem install redis

src/redis-trib.rb create --replicas 1 192.168.100.100:7001 192.168.100.100:7002 192.168.100.100:7003 192.168.100.100:7004 192.168.100.100:7005 192.168.100.100:7006

### 第八步：连接redis客户端

node01执行以下命令连接redis客户端

cd /export/redis-3.2.8

src/redis-cli -h node01 -c -p 7001

## 3、redis集群管理

添加一个新节点作为主节点

启动新节点的redis服务，然后添加到集群当中去

启动服务

src/redis-server clusters/7007/redis.conf

src/redis-trib.rb add-node 192.168.100.100:7007 192.168.100.100:7001

添加一个新节点作为副本

src/redis-server clusters/7008/redis.conf

src/redis-trib.rb add-node --slave 192.168.100.100:7008 192.168.100.100:7001

删除一个节点

命令格式

src/redis-trib del-node 127.0.0.1:7000 `<node-id>`

src/redis-trib.rb del-node 192.168.100.100:7008 7c7b7f68bc56bf24cbb36b599d2e2d97b26c5540

重新分片

./redis-trib.rb reshard node01:7001

./redis-trib.rb reshard --from <node-id> --to <node-id> --slots <number of slots> --yes <host>:<port>

## 4、JavaAPI操作redis集群

@Test  
**public void** jedisClusterTest() **throws** IOException {  
 Set<HostAndPort> portSet = **new** HashSet<HostAndPort>();  
 portSet.add(**new** HostAndPort(**"node01"**,7001));  
 portSet.add(**new** HostAndPort(**"node01"**,7002));  
 portSet.add(**new** HostAndPort(**"node01"**,7003));  
 portSet.add(**new** HostAndPort(**"node01"**,7004));  
 portSet.add(**new** HostAndPort(**"node01"**,7005));  
 portSet.add(**new** HostAndPort(**"node01"**,7006));  
 JedisPoolConfig jedisPoolConfig = **new** JedisPoolConfig();  
 jedisPoolConfig.setMaxIdle(10);  
 jedisPoolConfig.setMaxWaitMillis(3000);  
 jedisPoolConfig.setMaxTotal(30);  
 jedisPoolConfig.setMinIdle(5);  
  
 JedisCluster jedisCluster = **new** JedisCluster(portSet, jedisPoolConfig);  
  
 jedisCluster.set(**"cluster"**,**"clustervalue"**);  
  
 jedisCluster.close();  
  
}