**畅购电商系统开发**

**第16天**

传智播客.黑马程序员.深圳

# 学习目标

* 理解集群流程
* 理解分布式概念
* 能实现Eureka集群[集群配置]
* 能实现Redis集群[Redis集群配置、哨兵策略、Redis击穿、雪崩问题]
* RabbitMQ集群

# 集群概述

## 什么是集群

### 集群概念

集群是一种计算机系统， 它通过一组松散集成的计算机软件和/或硬件连接起来高度紧密地协作完成计算工作。在某种意义上，他们可以被看作是一台计算机。集群系统中的单个计算机通常称为节点，通常通过局域网连接，但也有其它的可能连接方式。集群计算机通常用来改进单个计算机的计算速度和/或可靠性。一般情况下集群计算机比单个计算机，比如工作站或超级计算机性能价格比要高得多。

### 集群的特点

集群拥有以下两个特点：

1. 可扩展性：集群的性能不限制于单一的服务实体，新的服务实体可以动态的添加到集群，从而增强集群的性能。
2. 高可用性：集群当其中一个节点发生故障时，这台节点上面所运行的应用程序将在另一台节点被自动接管，消除单点故障对于增强数据可用性、可达性和可靠性是非常重要的。

### 集群的两大能力

集群必须拥有以下两大能力：

1. 负载均衡：负载均衡把任务比较均匀的分布到集群环境下的计算和网络资源，以提高数据吞吐量。
2. 错误恢复：如果集群中的某一台服务器由于故障或者维护需要无法使用，资源和应用程序将转移到可用的集群节点上。这种由于某个节点的资源不能工作，另一个可用节点中的资源能够透明的接管并继续完成任务的过程，叫做错误恢复。

负载均衡和错误恢复要求各服务实体中有执行同一任务的资源存在，而且对于同一任务的各个资源来说，执行任务所需的信息视图必须是相同的。

## 集群与分布式的区别

说到集群，可能大家会立刻联想到另一个和它很相近的一个词----“分布式”。那么集群和分布式是一回事吗？有什么联系和区别呢?

**相同点：**

分布式和集群都是需要有很多节点服务器通过网络协同工作完成整体的任务目标。

**不同点：**

分布式是指将业务系统进行拆分，即分布式的每一个节点都是实现不同的功能。而集群每个节点做的是同一件事情。

如下图，每个人都有不同的分工，一起协作干一件事，叫做“分布式”



再看下图：每个划桨人干的都是一样的活，叫做集群。



分布式的每一个节点也可以做成集群。其实这个赛龙舟的图，总整体来看属于分布式,包括打鼓和划桨两个分布式节点，而划桨的节点又是集群的形态。

现实生活中例子还有很多，例如，这样古代乐队的图就属于集群



而现代乐队这样图就是分布式啦



# Eureka集群

## Eureka简介

### 什么是Eureka

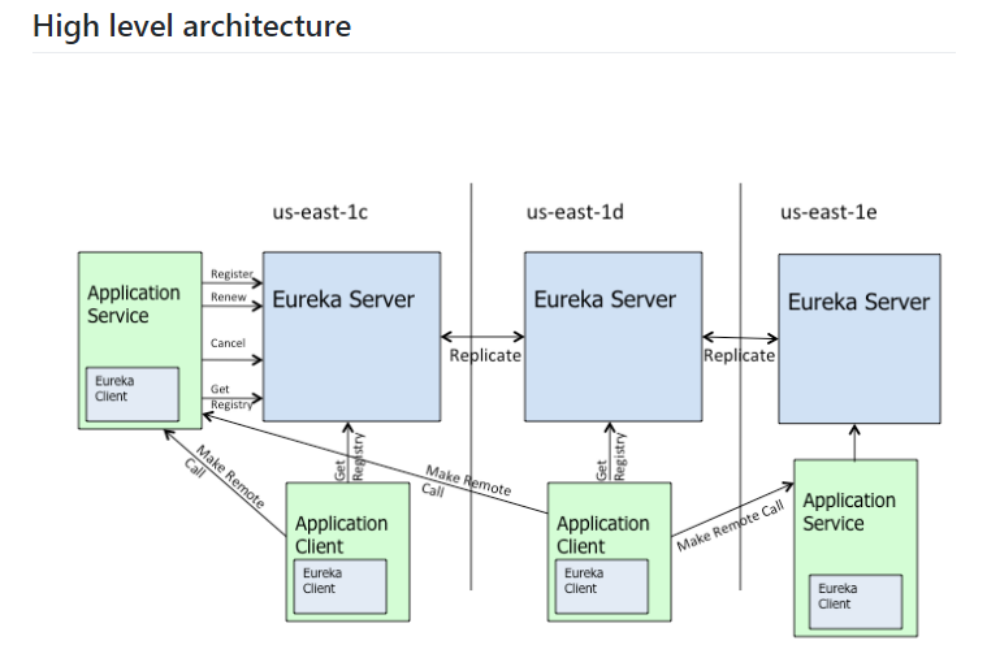
​ Eureka是一种基于REST（Representational State Transfer）的服务，主要用于AWS，用于定位服务，以实现中间层服务器的负载平衡和故障转移。我们将此服务称为Eureka Server。Eureka还附带了一个基于Java的客户端组件Eureka Client，它使与服务的交互变得更加容易。客户端还有一个内置的负载均衡器，可以进行基本的循环负载均衡。在Netflix，一个更复杂的负载均衡器包装Eureka，根据流量，资源使用，错误条件等多种因素提供加权负载平衡，以提供卓越的弹性。

理解：

​Eureka是一个服务注册与发现的注册中心。类似于dubbo中的zookeeper.

官网地址：<https://github.com/Netflix/eureka/wiki/Eureka-at-a-glance>

### Eureka的架构



application Service :相当于服务提供者

application Client :相当于服务消费者

make remote call :服务调用过程

us-east-1c d e 都是region:us-east-1 的可用区域。

简单可以理解为：

​ 每一个erurak都是一个节点，默认启动时就是以集群的方式。

## 搭建Eureka集群

配置host文件C:\Windows\System32\drivers\etc\hosts文件，添加映射

127.0.0.1 eureka-server1

127.0.0.1 eureka-server2

127.0.0.1 eureka-server3

### 父工程

创建eureka\_cluster工程用于模拟集群搭建，项目pom依赖如下：

*<?***xml version="1.0" encoding="UTF-8"***?>*<**project xmlns="http://maven.apache.org/POM/4.0.0"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"**>  
 <**modelVersion**>4.0.0</**modelVersion**>  
  
 <**groupId**>com.itheima</**groupId**>  
 <**artifactId**>eureka\_cluster</**artifactId**>  
 <**packaging**>pom</**packaging**>  
 <**version**>1.0-SNAPSHOT</**version**>  
  
 <**parent**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-parent</**artifactId**>  
 <**version**>2.1.7.RELEASE</**version**>  
 </**parent**>  
  
 <**dependencies**>  
 <**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-web</**artifactId**>  
 </**dependency**>  
  
 <**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-netflix-eureka-server</**artifactId**>  
 <**version**>2.1.1.RELEASE</**version**>  
 </**dependency**>  
 </**dependencies**>  
</**project**>

### 搭建集群子工程

创建三台Eureka服务器eureka\_server1、eureka\_server2、eureka\_server3工程并配置它们，同时添加启动类

**第1台application.yml：**

**server**:  
 **port**: 8761  
**eureka**:  
 **instance**:  
 *#指定hostname为域名-在本地hosts文件中有配置* **hostname**: eureka-server1  
 **client**:  
 **register-with-eureka**: **false  
 fetch-registry**: **false  
 service-url**:  
 *#配置整个集群中其它节点的信息* **defaultZone**: http://eureka-server2:8762/eureka/,http://eureka-server3:8763/eureka/

**第2台application.yml：**

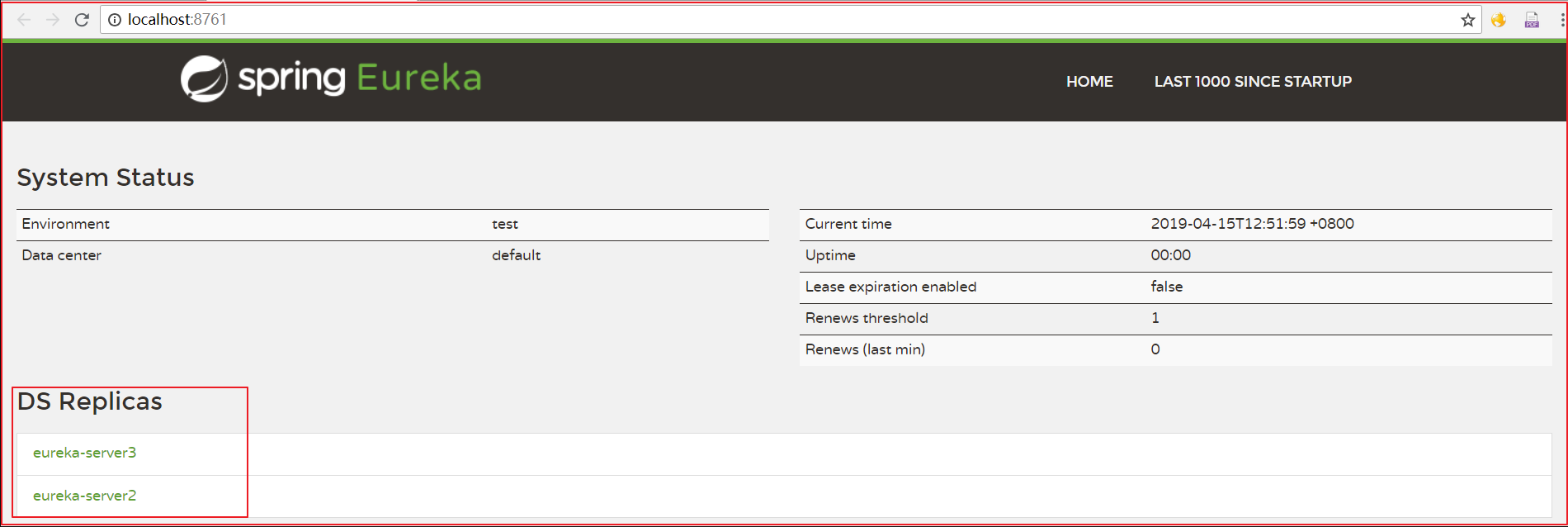
**server**:  
 **port**: 8762  
**eureka**:  
 **instance**:  
 *#指定hostname为域名-在本地hosts文件中有配置* **hostname**: eureka-server2  
 **client**:  
 **register-with-eureka**: **false  
 fetch-registry**: **false  
 service-url**:  
 *#配置整个集群中其它节点的信息* **defaultZone**: http://eureka-server1:8761/eureka/,http://eureka-server3:8763/eureka/

**第3台application.yml：**

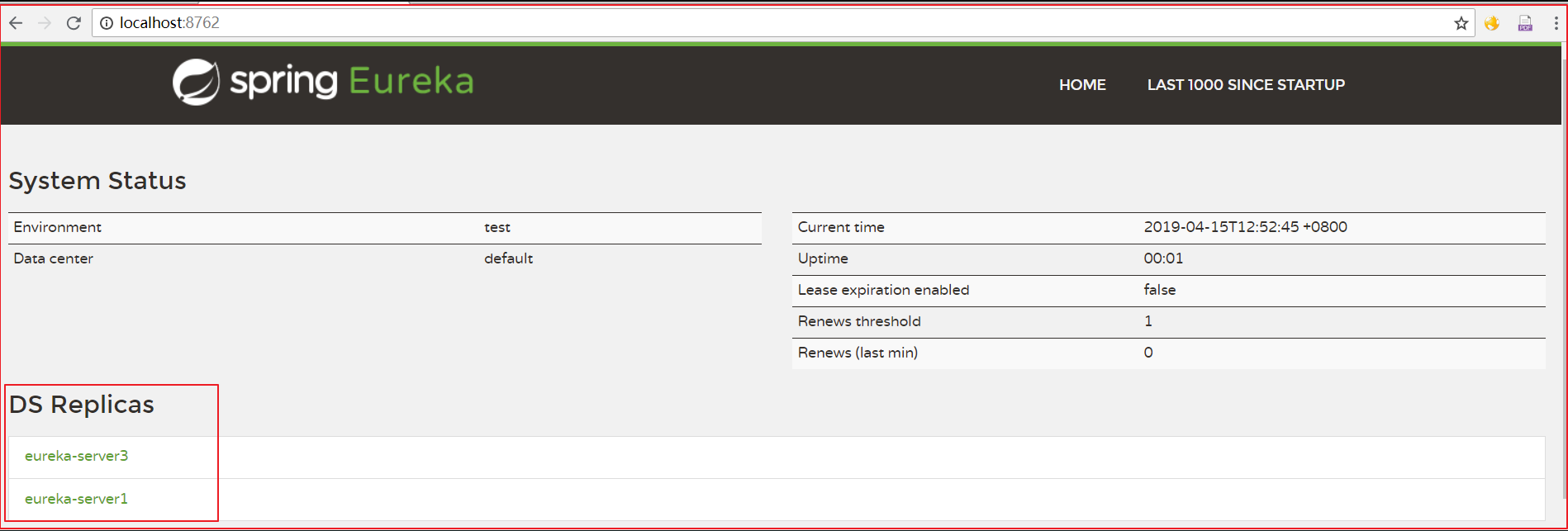
**server**:  
 **port**: 8763  
**eureka**:  
 **instance**:  
 *#指定hostname为域名-在本地hosts文件中有配置* **hostname**: eureka-server3  
 **client**:  
 **register-with-eureka**: **false  
 fetch-registry**: **false  
 service-url**:  
 *#配置整个集群中其它节点的信息* **defaultZone**: http://eureka-server2:8762/eureka/,http://eureka-server1:8761/eureka/

### 测试

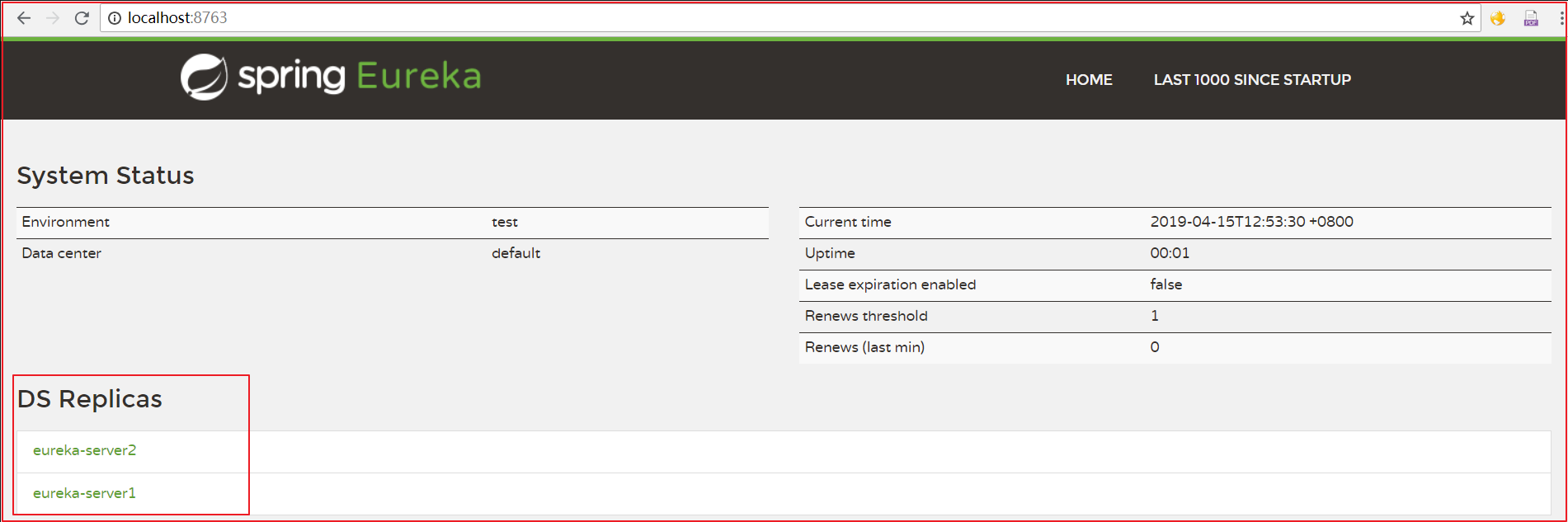
<http://localhost:8761/>



<http://localhost:8762/>



<http://localhost:8763/>



注意：此时如果我们的Eureka客户端想连接集群版本的eureka-server只需要在yml配置文件中：

**eureka**:  
 **client**:  
 **service-url**:  
 **defaultZone**: http://eureka-server1:8761/eureka/,http://eureka-server2:8762/eureka/,http://eureka-server3:8763/eureka/  
 **instance**:  
 **prefer-ip-address**: **true // prefer-ip-address设置为true以IP地址注册到服务中心**，

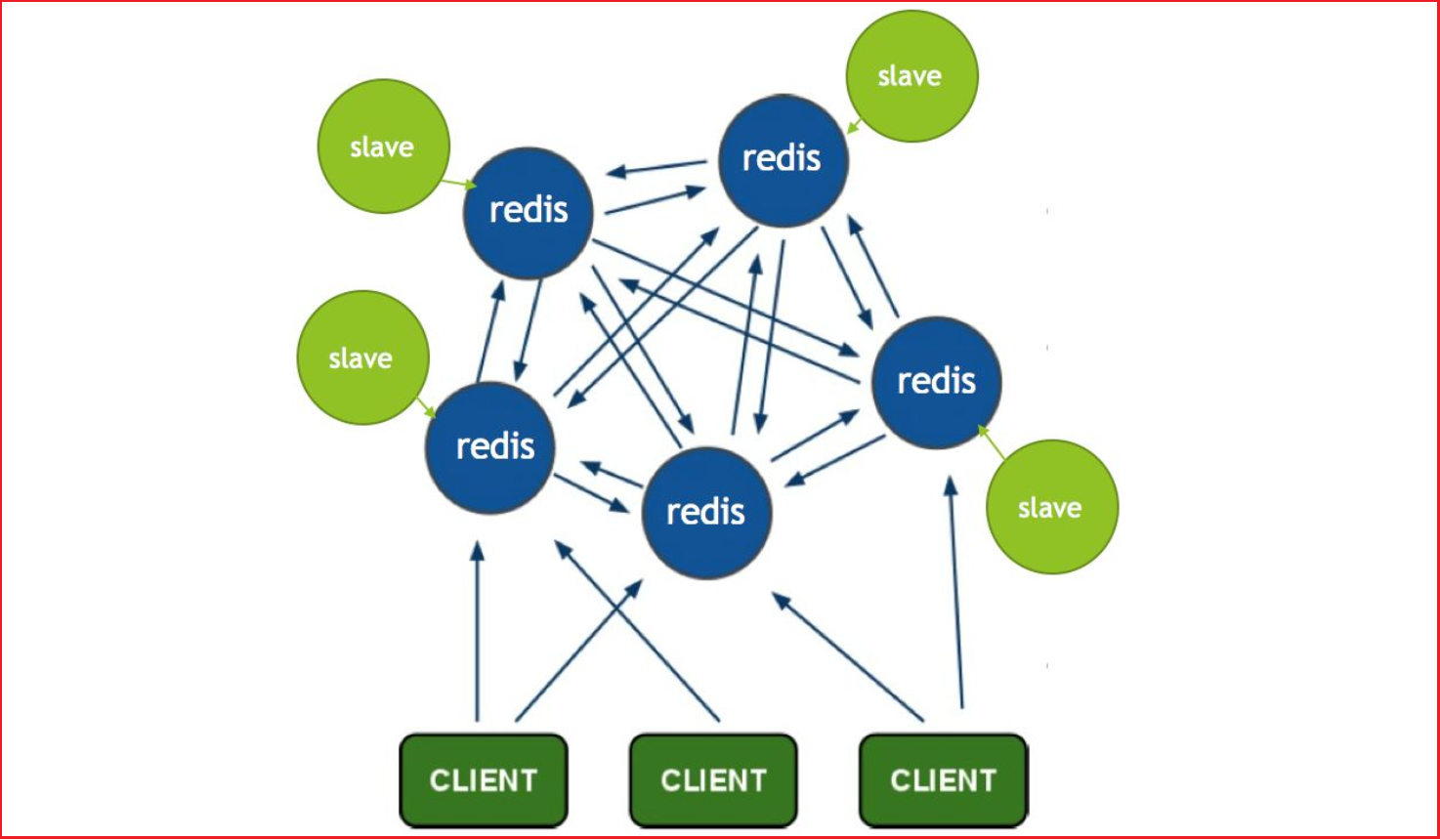
# Redis Cluster

## Redis-Cluster简介

### 什么是Redis-Cluster

为何要搭建Redis集群。Redis是在内存中保存数据的，而我们的电脑一般内存都不大，这也就意味着Redis不适合存储大数据，适合存储大数据的是Hadoop生态系统的Hbase或者是MogoDB。Redis更适合处理高并发，一台设备的存储能力是很有限的，但是多台设备协同合作，就可以让内存增大很多倍，这就需要用到集群。

Redis集群搭建的方式有多种，例如使用客户端分片、Twemproxy、Codis等，但从redis 3.0之后版本支持redis-cluster集群，它是Redis官方提出的解决方案，Redis-Cluster采用无中心结构，每个节点保存数据和整个集群状态,每个节点都和其他所有节点连接。其redis-cluster架构图如下：



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。

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.

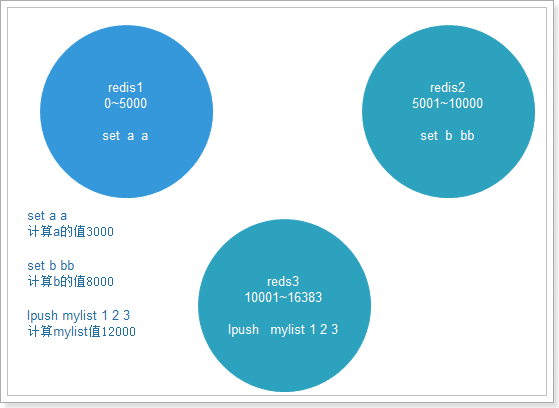
### 分布存储机制-槽

（1）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

（2）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。

例如三个节点：槽分布的值如下：



Redis可以无限扩展集群吗？当然不行，Redis最多集群16383台节点，只有这1万多个节点，够我们用吗？

号称全球最大的Redis集群维护商，新浪微博的工程师们曾经在多个公开场合都讲到过，微博平台当前在使用并维护着可能是世界上最大的Redis集群，其中最大的一个业务，单个业务使用了超过 10T 的内存，这里说的就是微博关系服务。

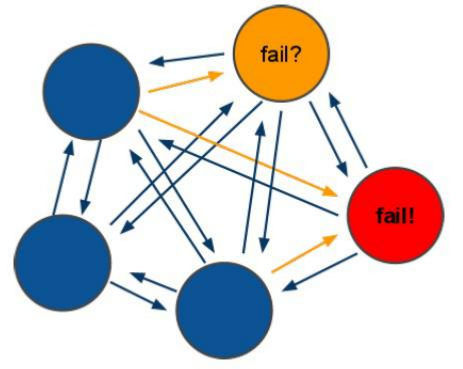
全球最大的Redis集群才用了10T数据，换算一下，按现在普通服务器都有50GB左右内存的机器，10T也就相当于: (10 \* 1024)/50 =204.8，差不多205台机器。

### 容错机制-投票

（1）选举过程是集群中所有master参与，如果半数以上master节点与故障节点通信超过(cluster-node-timeout)，认为该节点故障，自动触发故障转移操作。故障节点对应的从节点自动升级为主节点处理数据。

（2）什么时候整个集群不可用(cluster\_state:fail)?

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



## 搭建Redis-Cluster

### 搭建要求

需要 6 台 redis 服务器。搭建伪集群。

需要 6 个 redis 实例。

需要运行在不同的端口 7001-7006

### 准备工作

**（1）安装gcc 【此步省略】**

Redis 是 c 语言开发的。安装 redis 需要 c 语言的编译环境。如果没有 gcc 需要在线安装。

|  |
| --- |
| yum install gcc-c++ |

**（2）使用yum命令安装 ruby （我们需要使用ruby脚本来实现集群搭建）**

|  |
| --- |
| yum install ruby  yum install rubygems |

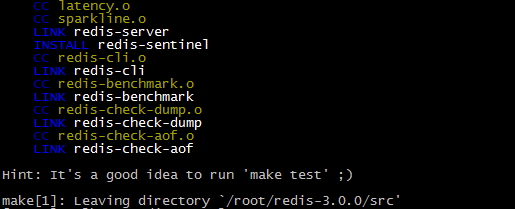
|  |
| --- |
| **----- 知识点小贴士 -----**  Ruby，一种简单快捷的[面向对象](https://baike.baidu.com/item/%E9%9D%A2%E5%90%91%E5%AF%B9%E8%B1%A1)（[面向对象程序设计](https://baike.baidu.com/item/%E9%9D%A2%E5%90%91%E5%AF%B9%E8%B1%A1%E7%A8%8B%E5%BA%8F%E8%AE%BE%E8%AE%A1)）[脚本语言](https://baike.baidu.com/item/%E8%84%9A%E6%9C%AC%E8%AF%AD%E8%A8%80)，在20世纪90年代由日本人松本行弘([Yukihiro Matsumoto](https://baike.baidu.com/item/Yukihiro%20Matsumoto))开发，遵守[GPL](https://baike.baidu.com/item/GPL)协议和Ruby License。它的灵感与特性来自于 [Perl](https://baike.baidu.com/item/Perl)、[Smalltalk](https://baike.baidu.com/item/Smalltalk)、[Eiffel](https://baike.baidu.com/item/Eiffel)、[Ada](https://baike.baidu.com/item/Ada/5606819)以及 [Lisp](https://baike.baidu.com/item/Lisp/22083) 语言。由 Ruby 语言本身还发展出了JRuby（[Java平台](https://baike.baidu.com/item/Java%E5%B9%B3%E5%8F%B0)）、IronRuby（[.NET](https://baike.baidu.com/item/.NET)平台）等其他平台的 Ruby 语言[替代品](https://baike.baidu.com/item/%E6%9B%BF%E4%BB%A3%E5%93%81)。Ruby的作者于1993年2月24日开始编写Ruby，直至1995年12月才正式[公开发布](https://baike.baidu.com/item/%E5%85%AC%E5%BC%80%E5%8F%91%E5%B8%83)于fj（[新闻组](https://baike.baidu.com/item/%E6%96%B0%E9%97%BB%E7%BB%84)）。因为Perl发音与6月诞生石pearl（[珍珠](https://baike.baidu.com/item/%E7%8F%8D%E7%8F%A0/33579)）相同，因此Ruby以7月诞生石ruby（[红宝石](https://baike.baidu.com/item/%E7%BA%A2%E5%AE%9D%E7%9F%B3/24242)）命名  RubyGems简称gems，是一个用于对 Ruby组件进行打包的 Ruby 打包系统 |

（3）将redis源码包上传到 linux 系统 ，解压redis源码包

（4）编译redis源码 ，进入redis源码文件夹

|  |
| --- |
| make |

看到以下输出结果，表示编译成功



（5）创建目录/usr/local/redis-cluster目录， 安装6个redis实例，分别安装在以下目录

/usr/local/redis-cluster/redis-1

/usr/local/redis-cluster/redis-2

/usr/local/redis-cluster/redis-3

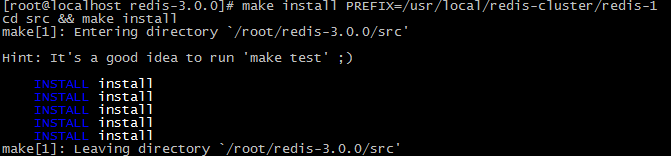
/usr/local/redis-cluster/redis-4

/usr/local/redis-cluster/redis-5

/usr/local/redis-cluster/redis-6

以第一个redis实例为例，命令如下

|  |
| --- |
| make install PREFIX=/usr/local/redis-cluster/redis-1 |



出现此提示表示成功，按此方法安装其余5个redis实例

（6）复制配置文件 将 /redis-3.0.0/redis.conf 复制到redis下的bin目录下

|  |
| --- |
| [root@localhost redis-3.0.0]# cp redis.conf /usr/local/redis-cluster/redis-1/bin  [root@localhost redis-3.0.0]# cp redis.conf /usr/local/redis-cluster/redis-2/bin  [root@localhost redis-3.0.0]# cp redis.conf /usr/local/redis-cluster/redis-3/bin  [root@localhost redis-3.0.0]# cp redis.conf /usr/local/redis-cluster/redis-4/bin  [root@localhost redis-3.0.0]# cp redis.conf /usr/local/redis-cluster/redis-5/bin  [root@localhost redis-3.0.0]# cp redis.conf /usr/local/redis-cluster/redis-6/bin |

### 配置集群

**（1）修改每个redis节点的配置文件redis.conf**

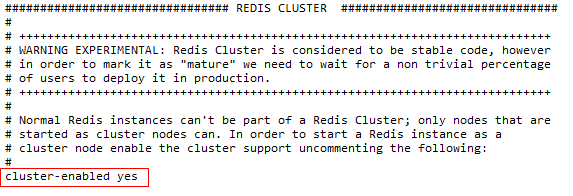
启动后台启动



修改运行端口为7001 （7002 7003 .....）



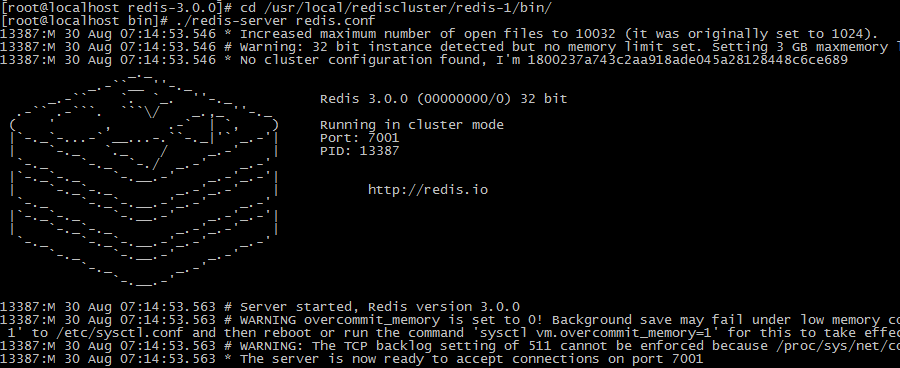
将cluster-enabled yes 前的注释去掉(632行)



**（2）启动每个redis实例**

以第一个实例为例，命令如下

|  |
| --- |
| cd /usr/local/redis-cluster/redis-1/bin/  ./redis-server redis.conf |



把其余的5个也启动起来，然后查看一下是不是都启动起来了

|  |
| --- |
| [root@localhost ~]# ps -ef | grep redis  root 15776 15775 0 08:19 pts/1 00:00:00 ./redis-server \*:7001 [cluster]  root 15810 15784 0 08:22 pts/2 00:00:00 ./redis-server \*:7002 [cluster]  root 15831 15813 0 08:23 pts/3 00:00:00 ./redis-server \*:7003 [cluster]  root 15852 15834 0 08:23 pts/4 00:00:00 ./redis-server \*:7004 [cluster]  root 15872 15856 0 08:24 pts/5 00:00:00 ./redis-server \*:7005 [cluster]  root 15891 15875 0 08:24 pts/6 00:00:00 ./redis-server \*:7006 [cluster]  root 15926 15895 0 08:24 pts/7 00:00:00 grep redis |

**（3）上传redis-3.0.0.gem ，安装 ruby用于搭建redis集群的脚本。**

|  |
| --- |
| [root@localhost ~]# gem install redis-3.0.0.gem  Successfully installed redis-3.0.0  1 gem installed  Installing ri documentation for redis-3.0.0...  Installing RDoc documentation for redis-3.0.0... |

**（4）使用 ruby 脚本搭建集群。**

进入redis源码目录中的src目录 执行下面的命令

|  |
| --- |
| ./redis-trib.rb create --replicas 1 192.168.211.132:7001 192.168.211.132:7002 192.168.211.132:7003 192.168.211.132:7004 192.168.211.132:7005 192.168.211.132:7006 |

出现下列提示信息

|  |
| --- |
| >>> Creating cluster  Connecting to node 192.168.211.132:7001: OK  Connecting to node 192.168.211.132:7002: OK  Connecting to node 192.168.211.132:7003: OK  Connecting to node 192.168.211.132:7004: OK  Connecting to node 192.168.211.132:7005: OK  Connecting to node 192.168.211.132:7006: OK  >>> Performing hash slots allocation on 6 nodes...  Using 3 masters:  192.168.211.132:7001  192.168.211.132:7002  192.168.211.132:7003  Adding replica 192.168.211.132:7004 to 192.168.211.132:7001  Adding replica 192.168.211.132:7005 to 192.168.211.132:7002  Adding replica 192.168.211.132:7006 to 192.168.211.132:7003  M: 1800237a743c2aa918ade045a28128448c6ce689 192.168.211.132:7001  slots:0-5460 (5461 slots) master  M: 7cb3f7d5c60bfbd3ab28800f8fd3bf6de005bf0d 192.168.211.132:7002  slots:5461-10922 (5462 slots) master  M: 436e88ec323a2f8bb08bf09f7df07cc7909fcf81 192.168.211.132:7003  slots:10923-16383 (5461 slots) master  S: c2a39a94b5f41532cd83bf6643e98fc277c2f441 192.168.211.132:7004  replicates 1800237a743c2aa918ade045a28128448c6ce689  S: b0e38d80273515c84b1a01820d8ecee04547d776 192.168.211.132:7005  replicates 7cb3f7d5c60bfbd3ab28800f8fd3bf6de005bf0d  S: 03bf6bd7e3e6eece5a02043224497c2c8e185132 192.168.211.132:7006  replicates 436e88ec323a2f8bb08bf09f7df07cc7909fcf81  Can I set the above configuration? (type 'yes' to accept): yes  >>> Nodes configuration updated  >>> Assign a different config epoch to each node  >>> Sending CLUSTER MEET messages to join the cluster  Waiting for the cluster to join....  >>> Performing Cluster Check (using node 192.168.211.132:7001)  M: 1800237a743c2aa918ade045a28128448c6ce689 192.168.211.132:7001  slots:0-5460 (5461 slots) master  M: 7cb3f7d5c60bfbd3ab28800f8fd3bf6de005bf0d 192.168.211.132:7002  slots:5461-10922 (5462 slots) master  M: 436e88ec323a2f8bb08bf09f7df07cc7909fcf81 192.168.211.132:7003  slots:10923-16383 (5461 slots) master  M: c2a39a94b5f41532cd83bf6643e98fc277c2f441 192.168.211.132:7004  slots: (0 slots) master  replicates 1800237a743c2aa918ade045a28128448c6ce689  M: b0e38d80273515c84b1a01820d8ecee04547d776 192.168.211.132:7005  slots: (0 slots) master  replicates 7cb3f7d5c60bfbd3ab28800f8fd3bf6de005bf0d  M: 03bf6bd7e3e6eece5a02043224497c2c8e185132 192.168.211.132:7006  slots: (0 slots) master  replicates 436e88ec323a2f8bb08bf09f7df07cc7909fcf81  [OK] All nodes agree about slots configuration.  >>> Check for open slots...  >>> Check slots coverage...  [OK] All 16384 slots covered. |

### 编写启动脚本

现在我们有6个redis实例，有些时测试时，我们需要重启所有实例，每个指令去执行太麻烦。

1. 在/usr/local/redis-cluster目录下vim startRedis.sh
2. 填入内容

cd /usr/local/redis-cluster/redis-1/bin/

./redis-server redis.conf

cd /usr/local/redis-cluster/redis-2/bin/

./redis-server redis.conf

cd /usr/local/redis-cluster/redis-3/bin/

./redis-server redis.conf

cd /usr/local/redis-cluster/redis-4/bin/

./redis-server redis.conf

cd /usr/local/redis-cluster/redis-5/bin/

./redis-server redis.conf

cd /usr/local/redis-cluster/redis-6/bin/

./redis-server redis.conf

1. 授权脚本可执行

chmod +x startRedis.sh

1. ./startRedis.sh

## 连接Redis-Cluster

### 客户端工具连接

Redis-cli 连接集群：

|  |
| --- |
| redis-cli -h 主机ip -p 端口（集群中任意端口） -c |

-c：代表连接的是 redis 集群

测试值的存取:

（1）从本地连接到集群redis 使用7001端口 加 -c 参数

（2）存入name值为abc ，系统提示此值被存入到了7002端口所在的redis （槽是5798）

（3）提取name的值，可以提取。

（4）退出（quit）

（5）再次以7001端口进入 ，不带-c

（6）查询name值，无法获取，因为值在7002端口的redis上

（7）我们以7002端口进入，获取name值发现是可以获取的,而以其它端口进入均不能获取

### springboot连接redis集群

#### 创建工程changgou\_redis\_demo，依赖如下：

*<?***xml version="1.0" encoding="UTF-8"***?>*<**project xmlns="http://maven.apache.org/POM/4.0.0"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"**>  
 <**modelVersion**>4.0.0</**modelVersion**>  
  
 <**groupId**>com.itheima</**groupId**>  
 <**artifactId**>changgou\_redis\_demo</**artifactId**>  
 <**version**>1.0-SNAPSHOT</**version**>  
  
 <**parent**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-parent</**artifactId**>  
 <**version**>2.1.7.RELEASE</**version**>  
 </**parent**>  
  
 <**dependencies**>  
 <**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-data-redis</**artifactId**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-web</**artifactId**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>org.mybatis.spring.boot</**groupId**>  
 <**artifactId**>mybatis-spring-boot-starter</**artifactId**>  
 <**version**>2.0.1</**version**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>mysql</**groupId**>  
 <**artifactId**>mysql-connector-java</**artifactId**>  
 <**scope**>runtime</**scope**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-test</**artifactId**>  
 <**scope**>test</**scope**>  
 </**dependency**>  
 </**dependencies**>  
</**project**>

#### 配置application.yml

**spring**:  
 **datasource**:  
 **driver-class-name**: com.mysql.jdbc.Driver  
 **url**: jdbc:mysql://127.0.0.1:3306/changgou\_user?useUnicode=true&characterEncoding=UTF-8&serverTimezone=UTC  
 **username**: root  
 **password**: itcast  
 **application**:  
 **name**: redis-demo  
 *#redis配置* **redis**:  
 **cluster**:  
 **nodes**:  
 - 192.168.211.132:7001  
 - 192.168.211.132:7002  
 - 192.168.211.132:7003  
 - 192.168.211.132:7004  
 - 192.168.211.132:7005  
 - 192.168.211.132:7006  
**server**:  
 **ssl**:  
 **enabled**: **false  
 port**: 9008  
**mybatis**:  
 **configuration**:  
 **map-underscore-to-camel-case**: **true**

#### 创建启动类与测试类进行测试

**启动类**

@SpringBootApplication  
**public class** RedisDemoApplication {  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(RedisDemoApplication.**class**, args);  
 }  
}

**测试类**

@RunWith(SpringRunner.**class**)  
@SpringBootTest  
**public class** RedisDemoApplicationTests {  
 @Autowired  
 **private** RedisTemplate **redisTemplate**;  
 @Test  
 **public void** contextLoads() {  
 **redisTemplate**.boundValueOps(**"name"**).set(**"Steven"**);  
 System.***out***.println(**redisTemplate**.boundValueOps(**"name"**).get());  
 }  
}

## Redis的持久化

### redis的持久化介绍

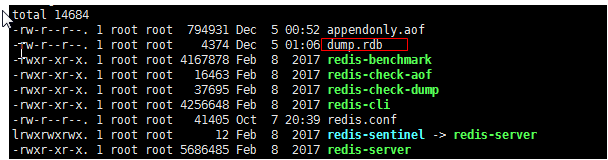
​ Redis的数据都放在内存中。如果机器挂掉，内存的数据就不存在，数据不能恢复，严重影响使用。那么redis本身给我们提供了持久化机制。即时出现这样的问题，也能恢复数据。接下来我们来看下redis的两种持久化方案分别为：RDB与AOF。

### 开启RDB

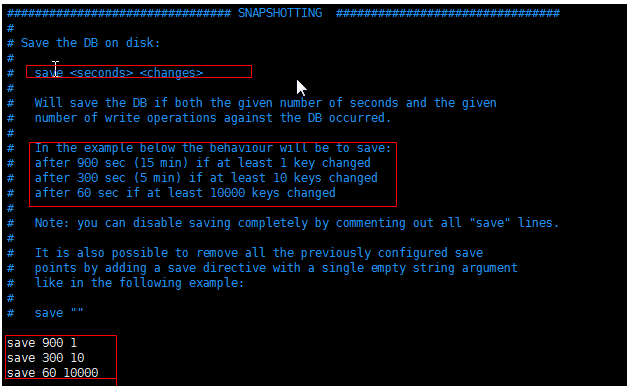
RDB: 快照形式 （定期数据保存磁盘中）会产生一个dump.rdb文件,redis默认开启了RDB的持久化方式。

特点：会存在数据丢失，性能较好，用于数据备份。

如图：有一个文件产生



如图：redis.conf中 的默认的RDB的配置：



解释：

# 在 900 秒内最少有 1 个 key 被改动，或者 300 秒内最少有 10 个 key 被改动，又或者 60 秒内最少有 1000 个 key 被改动，以上三个条件随便满足一个，就触发一次保存操作。

# if(在60秒之内有10000个keys发生变化时){

# 进行镜像备份

# }else if(在300秒之内有10个keys发生了变化){

# 进行镜像备份

# }else if(在900秒之内有1个keys发生了变化){

# 进行镜像备份

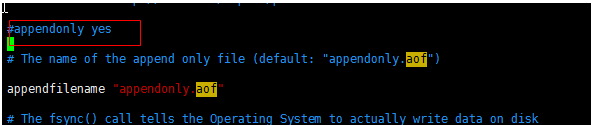
# }

### 开启AOF

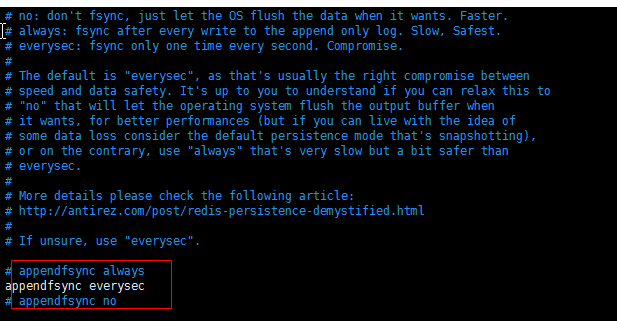
AOF (append only file)所有对redis的操作命令记录在.aof文件中,如果想恢复数据，重新加载文件，执行文件中的命令即可。默认的情况下 redis没有开启，要想开启，必须配置。

特点：每秒保存，数据完整性比较好，耗费性能。

开启AOF: 如图 去掉注释



配置 AOF的执行策略:

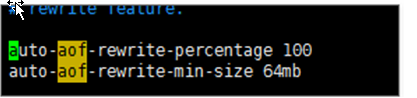


always:总是执行

everysec:每秒钟执行(默认)

no:不执行。

如果随着时间的推移，AOF文件中的数据越来越大，所以需要进行重写也就是压缩。



如图所示：自动压缩的比例为：

100：上一次AOF文件达到100%的时候进行压缩

64mb ：压缩时最小的文件大小。

### 模式的抉择应用场景介绍

AOF 和RDB对比：

| **命令** | **RDB** | **AOF** |
| --- | --- | --- |
| 启动优先级 | 低 | 高 |
| 体积 | 小 | 大 |
| 恢复速度 | 快 | 慢 |
| 数据安全性 | 丢数据 | 根据策略决定 |

RDB的最佳策略：

* 关闭
* 集中管理（用于备份数据）
* 主从模式，从开启。

AOF的最佳策略：

* 建议 开启每秒刷盘
* AOF重写集中管理(主从，从开启)

最佳的策略：

* 小分片（max\_memery 4G左右）
* 监控机器的负载

## Redis哨兵模式

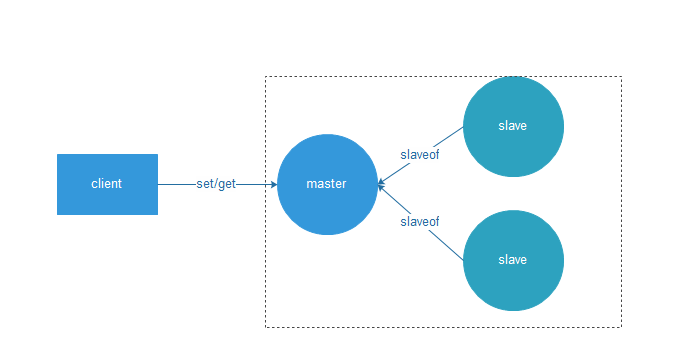
Redis在使用过程中服务器毫无征兆的宕机，是一个麻烦的事情，如何保证备份的机器是原始服务器的完整备份呢？这时候就需要哨兵和复制。

Sentinel（哨兵）可以管理多个Redis服务器，它提供了监控，提醒以及自动的故障转移的功能。

Replication（复制）则是负责让一个Redis服务器可以配备多个备份的服务器。

Redis也是利用这两个功能来保证Redis的高可用的。

### Redis的主从复制实现高可用

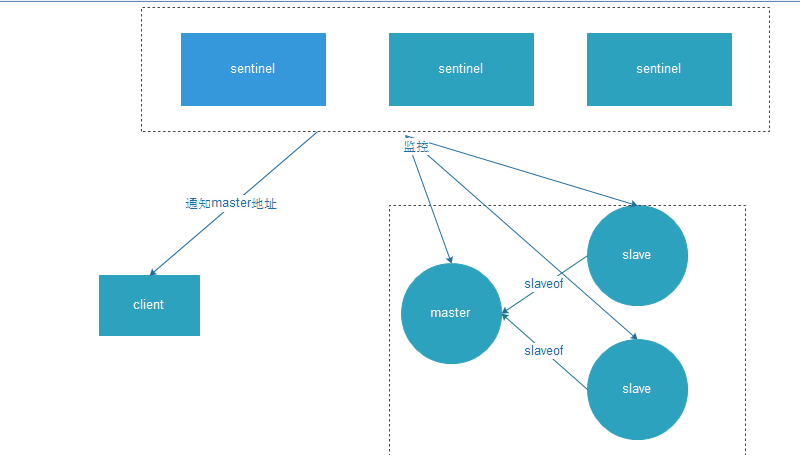


如图，通过主从架构，一个主节点，两个从节点。

通过手动监控的方式，监控master的宕机，以及出现故障将故障转移的方式可以做到高可用。

比如：如果主节点宕机，我们手动监控到主节点的宕机，并将某一个Slave变成主节点。 但是这样话，如何手动监控也是很麻烦的事情。所以使用sentinel机制就可以解决了这个问题，Sentinel（哨兵）是Redis 的高可用性解决方案。

* 它能自动进行故障转移。
* 客户端连接sentinel，不需要关注具体的master。
* 当master地址改变时由sentinel更新到客户端。



架构原理如图：

1.多个sentinel 发现并确认master有问题。

2.sentinel内部选举领导

3.选举出slave作为新的master

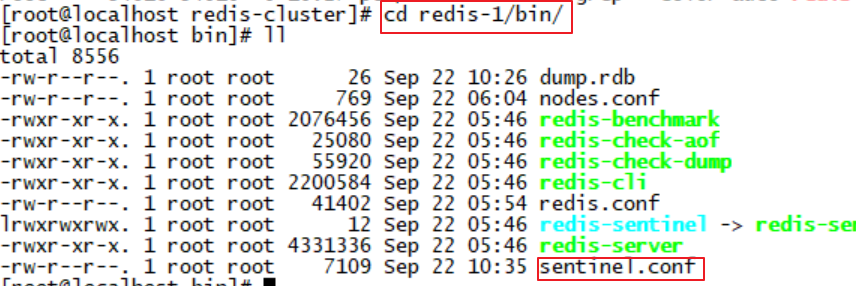
4.通知其余的slave成为新master的slave

5.通知客户端 主从变化

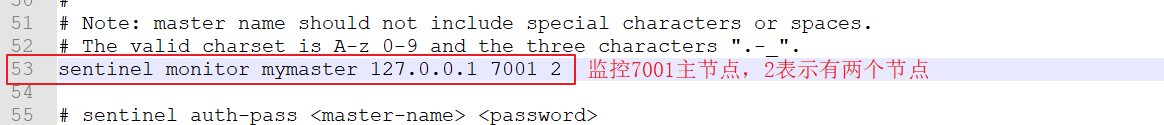
6.如果老的master重新复活，那么成为新的master的slave

### 配置哨兵

1. 把redis3.0.0解压包下的sentinel.conf复制到要监控的主节点bin目录中



1. 配置sentinel.conf



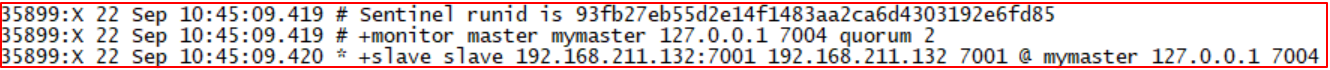
1. 杀死所有redis进程，并重启

pkill -9 redis

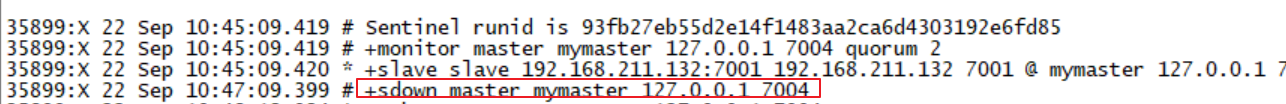
./startRedis.sh

1. 开启监听

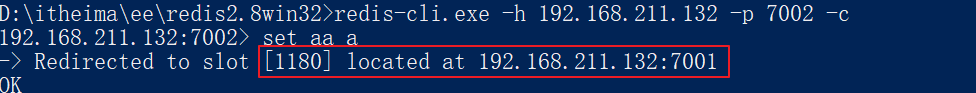
./redis-sentinel sentinel.conf



1. 杀死7004主节点，再次观看日志



1. 客户端重新连接redis-2,执行指令：set aa a



你会发现7001变成了主节点

要实现上边的功能的主要细节主要有以下三个定时任务：

**每10秒**，哨兵会向master和slave发送INFO命令(目的就是监控每一个节点信息)

**每2秒**，哨兵会向master库和slave的频道(\_\_sentinel\_\_:hello)发送自己的信息 （sentinel节点通过\_\_sentinel\_\_:hello频道进行信息交换，比如加入哨兵集群，分享自己节点数据）

**每1秒**，哨兵会向master和slave以及其他哨兵节点发送PING命令（目的就是 redis节点的状态监控，还有领导选举，主备切换选择等）

策略总结：

1. ​尽量为 每一个节点部署一个哨兵
2. ​哨兵也要搭建集群（防止哨兵单点故障）
3. ​每一个节点都同时设置quorum的值超过半数（N/2）+1

面试常问的问题：

​主从复制，以及哨兵 和集群之间区别。

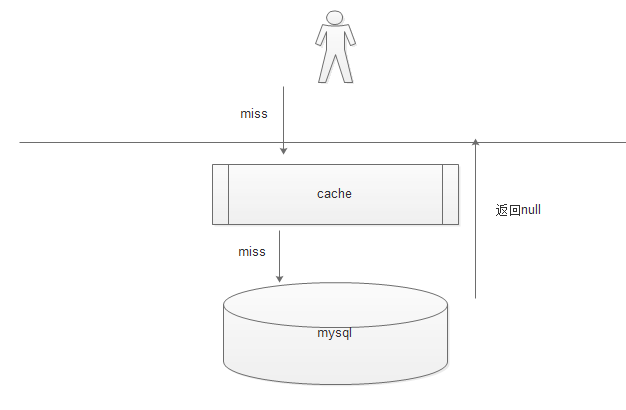
**主从复制：**redis实现高可用的一个策略。将会有主节点 和从节点，从节点的数据完整的从主节点中复制一份。

**哨兵：**当系统节点异常宕机的时候，开发者可以手动进行故障转移恢复，但是手动比较麻烦，所以通过哨兵机制自动进行监控和恢复。为了解决哨兵也会单点故障的问题，可以建立哨兵集群。

**集群：**即使使用哨兵，redis每个实例也是全量存储，每个redis存储的内容都是完整的数据，浪费内存且有木桶效应。为了最大化利用内存，可以采用集群，就是分布式存储。这个时候可以使用redis集群。将不同的数据分配到不同的节点中，这样就可以横向扩展，扩容。

## redis缓存击穿问题解决

### 什么是缓存击穿



如图：

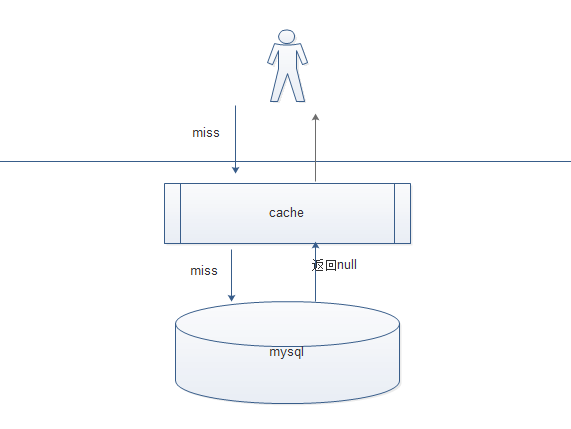
1. 当用户根据key 查询数据时，先查询缓存，如果缓存有命中，返回，

2. 但是如果缓存没有命中直接穿过缓存层，访问数据层 如果有，则存储指缓存，

3. 但是同样如果没有命中，（也就是数据库中也没有数据）直接返回用户，但是不缓存

这就是缓存的穿透。如果某一个key 请求量很大，但是存储层也没有数据，大量的请求都会达到存储层就会造成数据库压力巨大，有可能宕机的情况。

### 缓存击穿的解决方案



1.当缓存中没有命中的时候，从数据库中获取

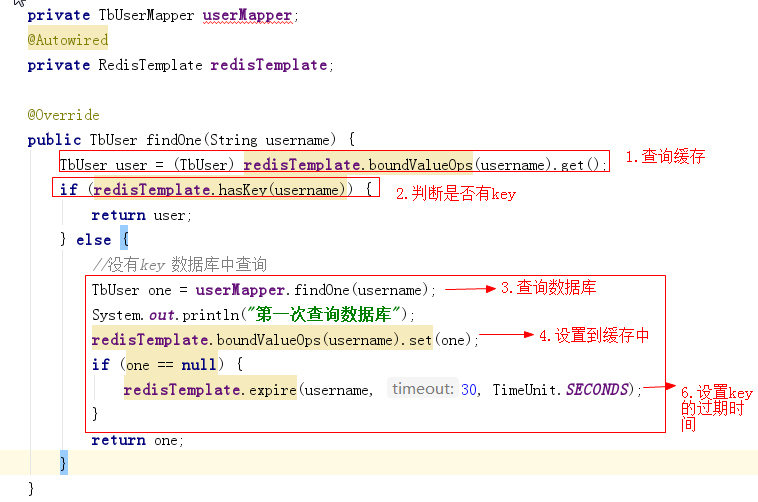
2.当数据库中也没有数据的时候，我们直接将null 作为值设置redis中的key上边。

3.此时如果没有数据，一般情况下都需要设置一个过期时间，例如：5分钟失效。（为了避免过多的KEY 存储在redis中）

4.返回给用户，

5.用户再次访问时，已经有KEY了。此时KEY的值是null而已，这样就可以在缓存中命中，解决了缓存穿透的问题。

例如：代码如下：



注意：缓存空对象会有两个问题：

第一，空值做了缓存，意味着缓存层中存了更多的键，需要更多的内存空间 ( 如果是攻击，问题更严重 )，比较有效的方法是针对这类数据设置一个较短的过期时间，让其自动剔除。

第二，缓存层和存储层的数据会有一段时间窗口的不一致，可能会对业务有一定影响。例如过期时间设置为 5分钟，如果此时存储层添加了这个数据，那此段时间就会出现缓存层和存储层数据的不一致，此时可以利用消息系统或者其他方式清除掉缓存层中的空对象。(此问题可以通过Canal解决)

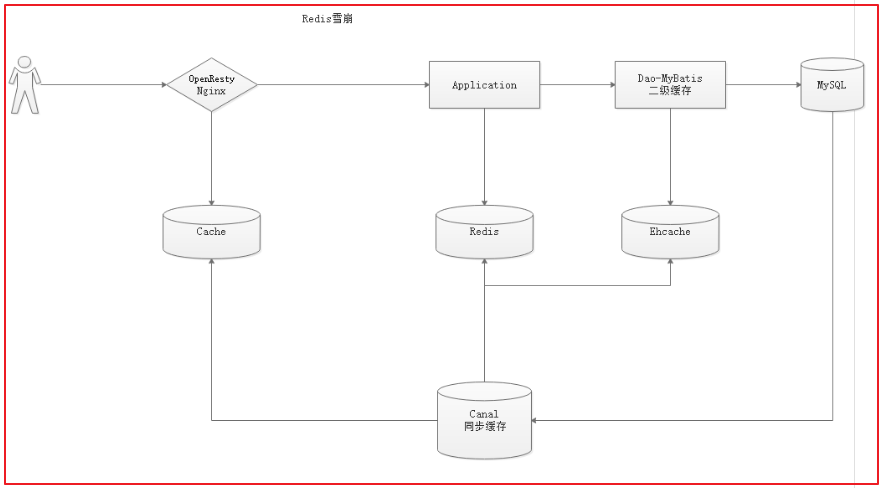
## Redis缓存雪崩问题解决(作业)

### 什么是缓存雪崩

如果缓存集中在一段时间内失效，发生大量的缓存穿透，所有的查询都落在数据库上，容易造成缓存雪崩。

### 如何解决

这个没有完美解决办法，但可以分析用户行为，尽量让失效时间点均匀分布。可以设计一套合理的缓存方案，如下：



#### 限流 加锁排队

在缓存失效后，通过对某一个key加锁或者是队列 来控制key的线程访问的数量。例如：某一个key 只允许一个线程进行 操作。

#### 限流

在缓存失效后，某一个key 做count统计限流，达到一定的阈值，直接丢弃，不再查询数据库。例如：令牌桶算法。等等。

#### 数据预热

在缓存失效应当尽量避免某一段时间，可以先进行数据预热，比如某些热门的商品。提前在上线之前，或者开放给用户使用之前，先进行loading 缓存中，这样用户使用的时候，直接从缓存中获取。要注意的是，要更加业务来进行过期时间的设置 ，尽量均匀。

#### 做缓存降级（二级缓存策略）

当分布式缓存失效的时候，可以采用本地缓存，本地缓存没有再查询数据库。这种方式，可以避免很多数据分布式缓存没有，就直接打到数据库的情况。

### 二级缓存解决雪崩的案例

分析：

​ 基本的思路：通过redis缓存+mybatis的二级缓存整合ehcache来实现。

（1）在原来的工程中加入依赖

<**dependency**>

<**groupId**>org.mybatis.caches</**groupId**>

<**artifactId**>mybatis-ehcache</**artifactId**>

<**version**>1.1.0</**version**>

</**dependency**>

（2）创建dao的接口 使用XML的方式

@Mapper

**public** **interface** **TbUserMapper** {

/\*\*

\* 根据用户名查询用户的信息

\* @param username

\* @return

\*/

**public** TbUser **findOne**(String username);

}

（3）创建TbUserMapper.xml，如图加入echache的配置 开启二级缓存

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE mapper

PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<**mapper** **namespace**="com.itheima.changgouredisdemo.dao.TbUserMapper">

<!--加入使用缓存-->

<**cache** **type**="org.mybatis.caches.ehcache.EhcacheCache">

<!--缓存自创建日期起至失效时的间隔时间一个小时-->

<**property** **name**="timeToIdleSeconds" **value**="3600"/>

<!--缓存创建以后，最后一次访问缓存的日期至失效之时的时间间隔一个小时-->

<**property** **name**="timeToLiveSeconds" **value**="3600"/>

<!--设置在缓存中保存的对象的最大的个数，这个按照业务进行配置-->

<**property** **name**="maxEntriesLocalHeap" **value**="1000"/>

<!--设置在磁盘中最大实体对象的个数-->

<**property** **name**="maxEntriesLocalDisk" **value**="10000000"/>

<!--缓存淘汰算法-->

<**property** **name**="memoryStoreEvictionPolicy" **value**="LRU"/>

</**cache**>

<**select** **id**="findOne" **resultType**="com.itheima.changgouredisdemo.pojo.TbUser" **parameterType**="string">

SELECT \* from tb\_user where username = #{username}

</**select**>

</**mapper**>

（4）配置application.yml

mybatis:

configuration:

map-underscore-to-camel-case: true

# 指定mapper映射文件目录

mapper-locations: classpath:mapper/\*Mapper.xml

（5）创建controller service 来进行测试：

**package** com.itheima.changgouredisdemo.controller;

**import** com.itheima.changgouredisdemo.pojo.TbUser;

**import** com.itheima.changgouredisdemo.service.UserService;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.web.bind.annotation.PathVariable;

**import** org.springframework.web.bind.annotation.RequestMapping;

**import** org.springframework.web.bind.annotation.RestController;

/\*\*

\* 描述

\*

\* @author 三国的包子

\* @version 1.0

\* @package com.itheima.changgouredisdemo.controller \*

\* @since 1.0

\*/

@RestController

@RequestMapping("/user")

**public** **class** **UserController** {

@Autowired

**private** UserService userService;

@RequestMapping("/findOne/{username}")

**public** TbUser **findOne**(@PathVariable String username) {

**return** userService.findOne(username);

}

}

**package** com.itheima.changgouredisdemo.service.impl;

**import** com.itheima.changgouredisdemo.dao.TbUserMapper;

**import** com.itheima.changgouredisdemo.pojo.TbUser;

**import** com.itheima.changgouredisdemo.service.UserService;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.data.redis.core.RedisTemplate;

**import** org.springframework.stereotype.Service;

**import** java.util.concurrent.TimeUnit;

/\*\*

\* 描述

\*

\* @author 三国的包子

\* @version 1.0

\* @package com.itheima.changgouredisdemo.service.impl \*

\* @since 1.0

\*/

@Service

**public** **class** **UserServiceImpl** **implements** **UserService** {

@Autowired

**private** TbUserMapper userMapper;

@Autowired

**private** RedisTemplate redisTemplate;

@Override

**public** TbUser **findOne**(String username) {

TbUser user = (TbUser) redisTemplate.boundValueOps(username).get();

**if** (redisTemplate.hasKey(username)) {

**return** user;

} **else** {

//没有key 数据库中查询

TbUser one = userMapper.findOne(username);

System.out.println("第一次查询数据库");

redisTemplate.boundValueOps(username).set(one);

**if** (one == **null**) {

redisTemplate.expire(username, 30, TimeUnit.SECONDS);

}

**return** one;

}

}

}

(6)测试：

已知： 数据库中有zhangsan

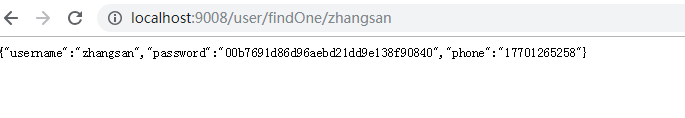


准备好redis

浏览器输入

http://localhost:9008/user/findOne/zhangsan

效果：



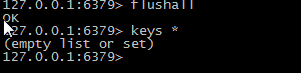
redis中: 也有数据



此时：修改数据库的数据zhangsan为zhangsan5,并清空redis缓存。

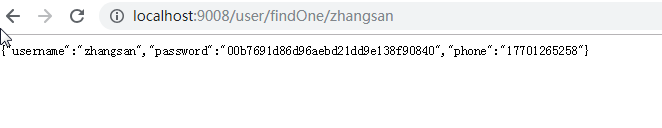
此时数据库总已经没有zhangsan,但是效果却是





再次输入浏览器地址：

http://localhost:9008/user/findOne/zhangsan



说明数据从二级缓存中也就是本地缓存中获取到了，测试成功。

# RabbitMQ集群

在使用RabbitMQ的过程中，如果只有一个节点，但是一旦单机版宕机，服务不可用，影响比较严重，所以这里我们演示下如何搭建rabbitmq的集群，集群就能避免单点故障的问题。

RabbitMQ 集群分为两种 普通集群 和 镜像集群

## 普通集群

以两个节点（rabbit01、rabbit02）为例来进行说明。

rabbit01和rabbit02两个节点仅有相同的元数据，即队列的结构，但消息实体只存在于其中一个节点rabbit01（或者rabbit02）中。

​ 当消息进入rabbit01节点的Queue后，consumer从rabbit02节点消费时，RabbitMQ会临时在rabbit01、rabbit02间进行消息传输，把A中的消息实体取出并经过B发送给consumer。

所以consumer应尽量连接每一个节点，从中取消息，即对于同一个逻辑队列，要在多个节点建立物理Queue；否则无论consumer连rabbit01或rabbit02，出口总在rabbit01，会产生瓶颈。

​ 当rabbit01节点故障后，rabbit02节点无法取到rabbit01节点中还未消费的消息实体。如果做了消息持久化，那么得等rabbit01节点恢复，然后才可被消费；如果没有持久化的话，就会产生消息丢失的现象。

## 镜像集群

​ 在普通集群的基础上，把需要的队列做成镜像队列，消息实体会主动在镜像节点间同步，而不是在客户端取数据时临时拉取，也就是说多少节点消息就会备份多少份。该模式带来的副作用也很明显，除了降低系统性能外，如果镜像队列数量过多，加之大量的消息进入，集群内部的网络带宽将会被这种同步通讯大大消耗掉。所以在对可靠性要求较高的场合中适用

由于镜像队列之间消息自动同步，且内部有选举master机制，即使master节点宕机也不会影响整个集群的使用，达到去中心化的目的，从而有效的防止消息丢失及服务不可用等问题。

## 集群搭建

准备两台虚拟机，虚拟机的地址分别为192.168.211.98, 192.168.211.99

Centos7

设置192.168.211.98的别名为A：修改如下文件 名称改为A

vi /etc/hostname

设置192.168.211.99的别名为B,修改如下文件 名称改为B

vi /etc/hostname

Centos6

需要修改两处：一处是/etc/sysconfig/network，另一处是/etc/hosts，只修改任一处会导致系统启动异常。首先切换到root用户。

**/etc/sysconfig/network**

打开该文件，里面有一行 `HOSTNAME=localhost.localdomain，修改 localhost.localdomain 为你的主机名。

**/etc/hosts**

打开该文件，会有一行 127.0.0.1 localhost.localdomain localhost其中 127.0.0.1 是本地环路地址， localhost.localdomain 是主机名(hostname)，也就是你待修改的。

修改每一个虚拟机的hosts

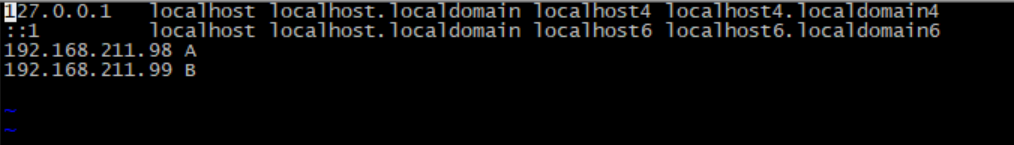
vi /etc/hosts

加入如下代码：

192.168.211.98 A

192.168.211.99 B

如下图所示：



重启系统。

### 安装erlang

两台机器上安装如下：

输入命令，下载erlang：

wget http://www.rabbitmq.com/releases/erlang/erlang-18.1-1.el6.x86\_64.rpm --no-check-certificate

安装erlang，root用户使用rpm安装：

rpm -ihv erlang-18.1-1.el6.x86\_64.rpm

### 下载和安装RabbitMQ

下载RabbitMQ:

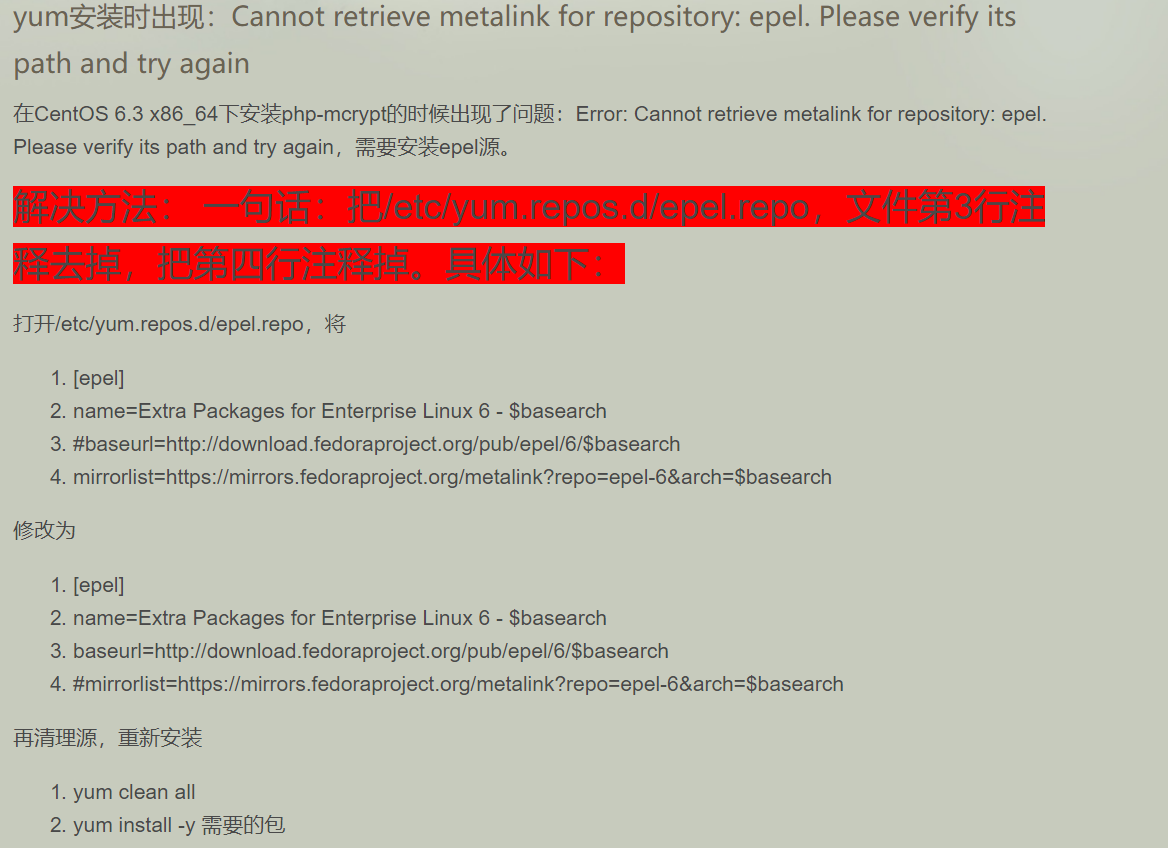
wget https://github.com/rabbitmq/rabbitmq-server/releases/download/rabbitmq\_v3\_6\_12/rabbitmq-server-3.6.12-1.el6.noarch.rpm

安装RabbitMQ,

1)先安装socat

yum -y install epel-release

yum install -y socat



2)root用户使用rpm安装 安装rabbitmq

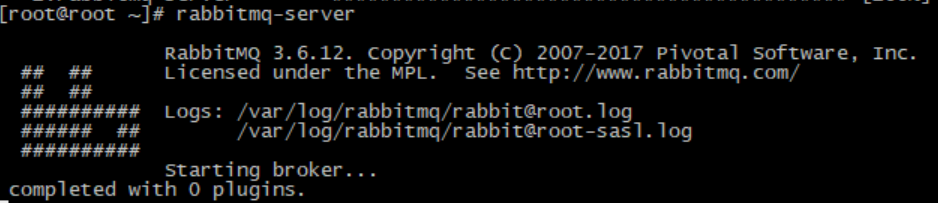
rpm -ihv rabbitmq-server-3.6.12-1.el6.noarch.rpm

验证是否安装成功：

输入命令：

[root@A ~]# rabbitmq-server

出现如下界面，说明OK



守护进程启动方式：

rabbitmq-server -detached

其他相关指令：

启动服务：rabbitmq-server -detached

查看状态：rabbitmqctl status

关闭服务：rabbitmqctl stop

列出角色：rabbitmqctl list\_users

改密码: rabbimqctlchange\_password {username} {newpassword}

删除用户: rabbitmqctl delete\_user xx

### 配置RabbitMQ

（1）创建rabbitmq的账号

[root@A ~]# rabbitmqctl add\_user admin admin

（2）将admin账号赋予管理员权限

[root@A ~]# rabbitmqctl set\_user\_tags admin administrator

（3）设置权限

[root@A ~]# rabbitmqctl set\_permissions -p '/' admin '.' '.' '.'

安装可视化界面

rabbitmq-plugins enable rabbitmq\_management

## 配置rabbitmq集群

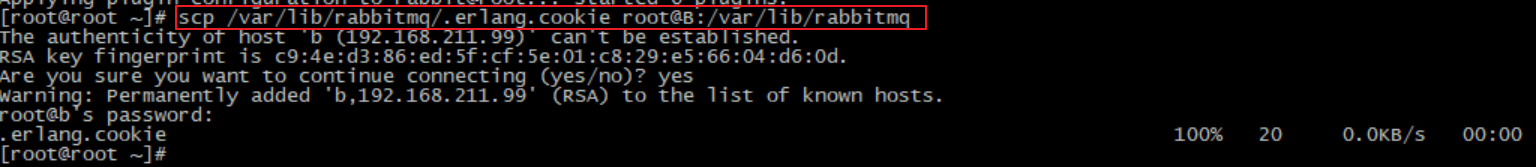
### 安装rabbitmq

安装方法如上边的方法进行在 192.168.211.99上进行

### 保证相同的erlang cookie

erlang.cookie是erlang分布式的token文件，集群内所有的设备要持有相同的.erlang.cookie文件才允许彼此通信。

scp /var/lib/rabbitmq/.erlang.cookie root@B:/var/lib/rabbitmq



### 运行各个rabbitmq节点

先停止每一个服务器的服务

输入命令停止

rabbitmqctl stop

再输入命令启动：

rabbitmq-server -detached

### 查看状态

[root@A rabbitmq]# rabbitmqctl cluster\_status

### 将节点连接成集群

每一个服务器都关闭防火墙：

[root@B ~]# systemctl stop firewalld

B加入到集群A中，在B中执行如下命令：

[root@B ~]# rabbitmqctl stop\_app

[root@B ~]# rabbitmqctl join\_cluster rabbit@A

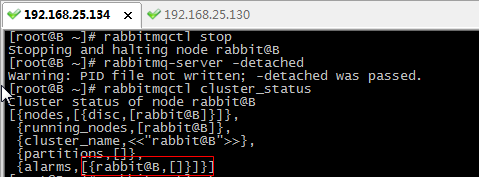
[root@B ~]# rabbitmqctl start\_app

A不需要加入到自己

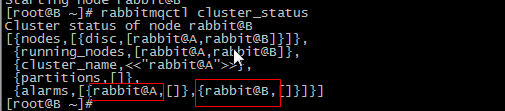
查看集群状态：在任意服务器上执行：

rabbitmqctl cluster\_status

添加集群之前：



添加集群之后：



### 配置单web插件查看集群

在A服务器执行命令：

rabbitmq-plugins enable rabbitmq\_management

在B服务器执行命令：

rabbitmq-plugins enable rabbitmq\_management

在浏览器中输入A服务器的rabbitmq服务器地址：

http://192.168.25.130:15672

输入用户名和密码登录，如果登录失败，采取如下措施：

* 执行命令 设置guest为管理员

rabbitmqctl **set**\_user\_tags guest administrator

* 执行命令 设置权限

rabbitmqctl **set**\_permissions -p / guest '.\*' '.\*' '.\*'

* 在rabbitmq的配置文件目录下创建一个rabbitmq.config文件

**cd** /etc/rabbitmq

**vi** rabbitmq.config

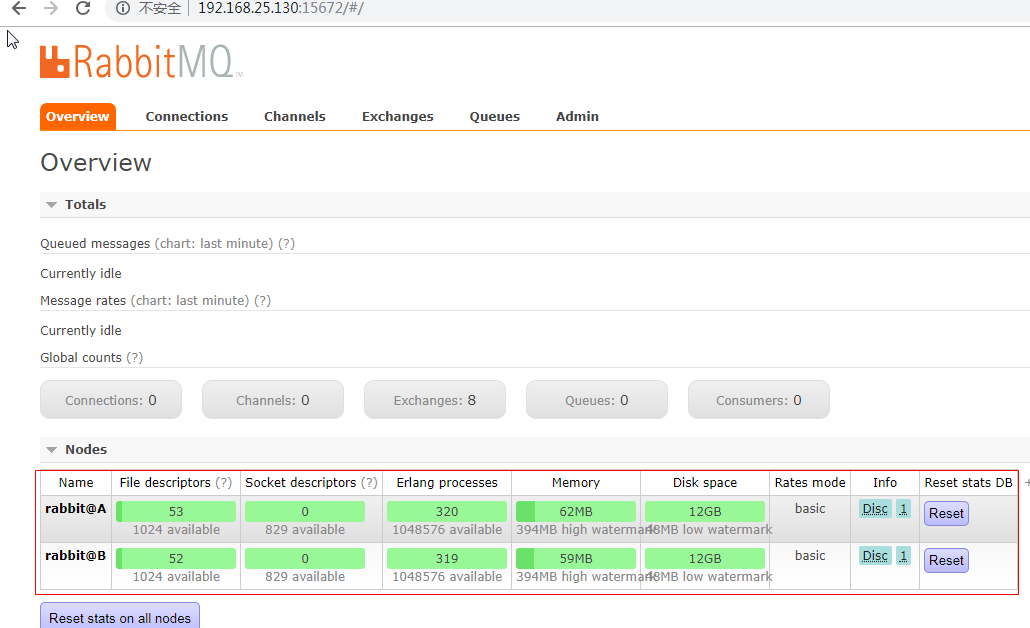
输入如下文本，并保存：

[{rabbit, [{loopback\_users, []}]}].

重新启动服务：

**service** **rabbitmq**-server restart

再次输入地址 出现界面，说明集群搭建成功。



如有需要，同理 对服务B的配置如上边一样配置一遍即可。

### 配置镜像队列

如上，我们已经搭建好了集群，但是并不能做到高可用，所以需要配置升级为镜像队列。

在任意的节点（A或者B）中执行如下命令：

rabbitmqctl set\_policy ha-all "^" '{"ha-mode":"all"}'

解释

rabbitmqctl set\_policy

用于设置策略

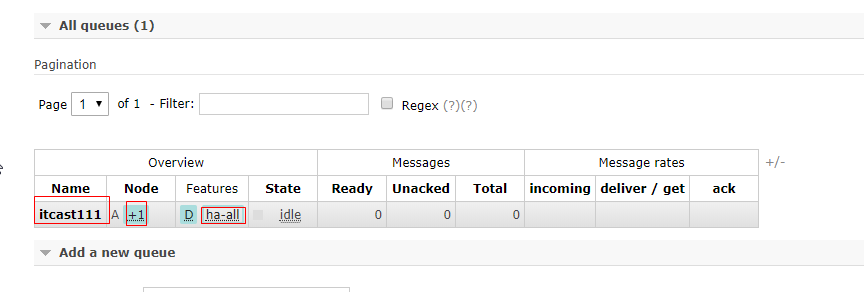
ha-all

表示设置为镜像队列并策略为所有节点可用 ，意味着 队列会被（同步）到所有的节点，当一个节点被加入到集群中时，也会同步到新的节点中，此策略比较保守，性能相对低，对接使用半数原则方式设置（N/2+1），例如：有3个结点 此时可以设置为：ha-two 表示同步到2个结点即可。

"^" 表示针对的队列的名称的正则表达式，此处表示匹配所有的队列名称

'{"ha-mode":"all"}' 设置一组key/value的JSON 设置为高可用模式 匹配所有exchange

此时查看web管理界面：添加一个队列itcast111,如下图已经可以出现结果为有一个结点，并且是ha-all模式（镜像队列模式）



集群最优解决方案参考: <https://www.cnblogs.com/xishuai/p/centos-rabbitmq-cluster-and-haproxy.html>

## springboot整合rabbitmq集群使用

修改原来redis测试的项目，

(1)加入pom.xml依赖

<**dependency**>

<**groupId**>org.springframework.boot</**groupId**>

<**artifactId**>spring-boot-starter-amqp</**artifactId**>

<**version**>2.1.4.RELEASE</**version**>

</**dependency**>

（2）配置application.yml

spring:

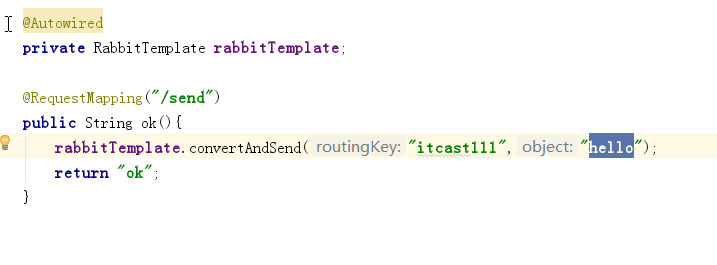
rabbitmq:

addresses: 192.168.25.130:5672,192.168.25.134:5672

username: guest

password: guest

(3)创建controller



**package** com.itheima.changgouredisdemo.controller;

**import** com.itheima.changgouredisdemo.pojo.TbUser;

**import** com.itheima.changgouredisdemo.service.UserService;

**import** org.springframework.amqp.rabbit.core.RabbitTemplate;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.web.bind.annotation.PathVariable;

**import** org.springframework.web.bind.annotation.RequestMapping;

**import** org.springframework.web.bind.annotation.RestController;

/\*\*

\* 描述

\*

\* @author 三国的包子

\* @version 1.0

\* @package com.itheima.changgouredisdemo.controller \*

\* @since 1.0

\*/

@RestController

@RequestMapping("/user")

**public** **class** **UserController** {

@Autowired

**private** UserService userService;

@RequestMapping("/findOne/{username}")

**public** TbUser **findOne**(@PathVariable String username) {

**return** userService.findOne(username);

}

@Autowired

**private** RabbitTemplate rabbitTemplate;

@RequestMapping("/send")

**public** String **ok**(){

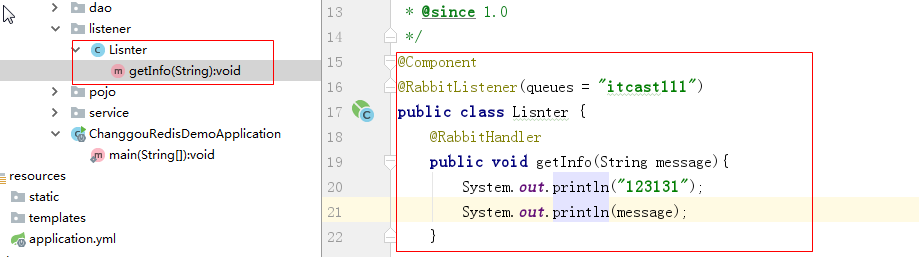
rabbitTemplate.convertAndSend("itcast111","hello");

**return** "ok";

}

}

(4)设置监听：



代码：

**package** com.itheima.changgouredisdemo.listener;

**import** org.springframework.amqp.rabbit.annotation.RabbitHandler;

**import** org.springframework.amqp.rabbit.annotation.RabbitListener;

**import** org.springframework.stereotype.Component;

/\*\*

\* 描述

\*

\* @author 三国的包子

\* @version 1.0

\* @package com.itheima.changgouredisdemo.listener \*

\* @since 1.0

\*/

@Component

@RabbitListener(queues = "itcast111")

**public** **class** **Lisnter** {

@RabbitHandler

**public** **void** **getInfo**(String message){

System.out.println("123131");

System.out.println(message);

}

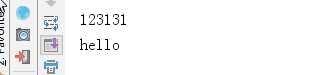
}

测试，在浏览器中输入：

**http**://localhost:9008/user/send

测试当宕机一台rabbitmq也能发送成功。

如下效果。



# 附录：Nginx+tomcat集群

==Nginx==：反向和正向代理服务器，轻量级的web服务器，也是邮件服务器。占用资源小，运行稳定。

命令：

启动命令：start nginx

停止命令：nginx -s quit

停止命令：nginx -s stop

重新加载：nginx -s reload

Nginx+Tomcat集群/高可用

#负载均衡池

**upstream** itheimaStream {

**server** 127.0.0.1:18081;

**server** 127.0.0.1:18082;

**server** 127.0.0.1:18083;

}

#配置虚拟机 处理www.itheima.com

**server** {

#侦听端口

**listen** 80;

#当前虚拟机所配置的域名信息[所有该域名访问都遵循该配置规则]

**server\_name** www.itheima.com;

#charset koi8-r;

#access\_log logs/host.access.log main;

#针对用户请求路径而言

#位置方位的意思

#根据用户的请求，决定如何路由用户的请求

#/所有请求

**location** / {

#所有www.itheima.com的请求都交给itheimaStream负载均衡池处理

#proxy\_pass http://itheimaStream;

**proxy\_pass** http://127.0.0.1:18081;

}

}

Nginx静态网站发布

#所有静态页访问

server {

#侦听端口

listen 80;

#当前虚拟机所配置的域名信息[所有该域名访问都遵循该配置规则]

server\_name resource.itheima.com;

#charset koi8-r;

#access\_log logs/host.access.log main;

#针对用户请求路径而言

#位置方位的意思

#根据用户的请求，决定如何路由用户的请求

#/所有请求

location / {

#所有的请求全部到D:\project-changgou-portal-fis3-master目录下找对应文件

root D:/project-changgou-portal-fis3-master;

}

}

Nginx实现动静分离

#配置虚拟机 处理www.itheima.com

**server** {

#侦听端口

**listen** 80;

#当前虚拟机所配置的域名信息[所有该域名访问都遵循该配置规则]

**server\_name** www.itheima.com;

#charset koi8-r;

#access\_log logs/host.access.log main;

#所有以.jpg，.png。。。都遵循该配置路由规则

**location** ~ \.(jpg|png|gif|js|css)$ {

#都直接到D盘的images目录找文件

**root** D:/images;

}

#针对用户请求路径而言

#位置方位的意思

#根据用户的请求，决定如何路由用户的请求

#/所有请求

**location** / {

#所有www.itheima.com的请求都交给itheimaStream负载均衡池处理

#proxy\_pass http://itheimaStream;

**proxy\_pass** http://127.0.0.1:18081;

}

}