# Redis简介

Redis 是完全开源免费的，遵守BSD协议，是一个高性能的key-value数据库。

Redis 与其他 key - value 缓存产品有以下三个特点：

1. Redis支持数据的持久化，可以将内存中的数据保存在磁盘中，重启的时候可以再次加载进行使用。
2. Redis不仅仅支持简单的key-value类型的数据，同时还提供list，set，zset，hash等数据结构的存储。
3. Redis支持数据的备份，即master-slave模式的数据备份

Redis优势：

1. 性能极高 – Redis能读的速度是110000次/s,写的速度是81000次/s 。
2. 丰富的数据类型 – Redis支持二进制案例的 Strings, Lists, Hashes, Sets 及 Ordered Sets 数据类型操作。
3. 原子 – Redis的所有操作都是原子性的，意思就是要么成功执行要么失败完全不执行。单个操作是原子性的。多个操作也支持事务，即原子性，通过MULTI和EXEC指令包起来。
4. 丰富的特性 – Redis还支持 publish/subscribe, 通知, key 过期等等特性。

Redis与其他key-value存储有什么不同？

1. Redis有着更为复杂的数据结构并且提供对他们的原子性操作，这是一个不同于其他数据库的进化路径。Redis的数据类型都是基于基本数据结构的同时对程序员透明，无需进行额外的抽象。
2. Redis运行在内存中但是可以持久化到磁盘，所以在对不同数据集进行高速读写时需要权衡内存，因为数据量不能大于硬件内存。在内存数据库方面的另一个优点是，相比在磁盘上相同的复杂的数据结构，在内存中操作起来非常简单，这样Redis可以做很多内部复杂性很强的事情。同时，在磁盘格式方面他们是紧凑的以追加的方式产生的，因为他们并不需要进行随机访问。

# Redis存储

Redis中数据存储模式有2种：cache-only,persistence;

cache-only即只做为“缓存”服务，不持久数据，数据在服务终止后将消失，此模式下也将不存在“数据恢复”的手段，是一种安全性低/效率高/容易扩展的方式；

persistence即为内存中的数据持久备份到磁盘文件，在服务重启后可以恢复，此模式下数据相对安全。

对于persistence持久化存储，Redis提供了两种持久化方法：

Redis DataBase(简称RDB)

Append-only file (简称AOF)

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**

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。**

# 单机版安装

安装redis需要先将官网下载的源码进行编译，编译依赖gcc环境，如果没有gcc环境，需要安装gcc：yum install make cmake gcc gcc-c++



把源码包上传到/usr/local/software/下

解压

tar -zxvf redis-3.2.11.tar.gz

进入解压后的目录，编译make



如果Redis 执行 make #error "Newer version of jemalloc required"解决办法是执行：

make MALLOC=libc

安装，PREFIX参数指定redis的安装目录，命令如下：

make install PREFIX=/usr/local/software/redis



进入安装后的路径，就是cd /usr/local/software/redis

在bin目录下执行

cd bin

./redis-server



这种方式是前端启动

缺点是ssh命令窗口关了之后服务就会停止

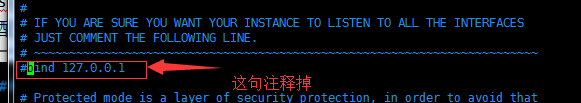
后台启动：

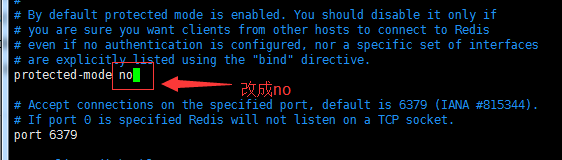
把redis源码包解压目录下的redis.conf复制到安装目录的bin下

cp /usr/local/software/redis-3.2.11/redis.conf /usr/local/software/redis/bin

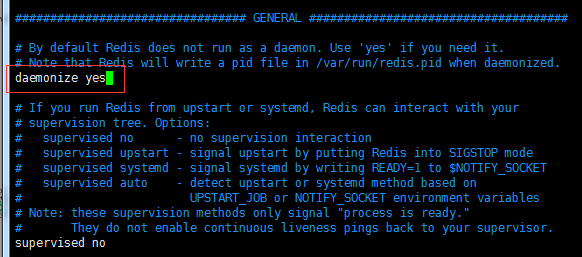
修改配置文件（bin下面的redis.conf）：

redis默认只能本机访问，要把下面两句改一下才能远程访问。





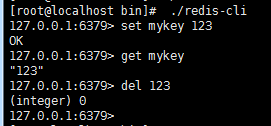
设置后台启动



在bin目录下执行./redis-server redis.conf

测试连接

./redis-cli



停止redis

./bin/redis-cli shutdown

建议直接kill，不然有可能关不掉

指定连接redis服务的ip和端口：

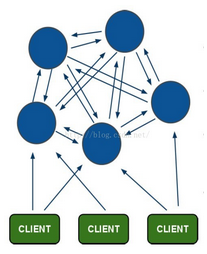


查看redis状态：

ps -ef | grep redis

# 集群搭建

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](http://lib.csdn.net/base/architecture)图如下：



架构细节:

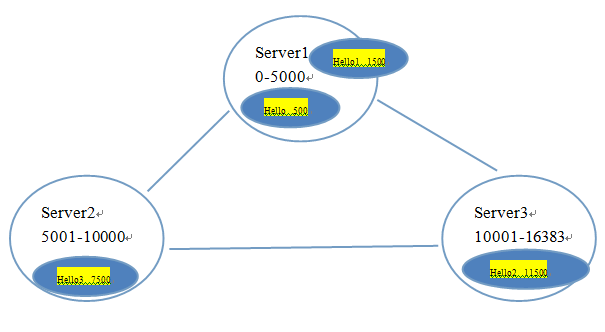
(1)所有的redis节点彼此互联(PING-PONG机制),内部使用二进制协议优化传输速度和带宽.

(2)节点的fail是通过集群中超过半数的节点检测失效时才生效.

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

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

Redis 集群中内置了 16384 个哈希槽，当需要在 Redis 集群中放置一个 key-value 时，redis 先对 key 使用 crc16 算法算出一个结果，然后把结果对 16384 求余数，这样每个 key 都会对应一个编号在 0-16383 之间的哈希槽，redis 会根据节点数量大致均等的将哈希槽映射到不同的节点。新增一个主节点，redis cluster的会从各个节点的前面各拿取一部分slot到新节点上。



## 安装ruby运行环境

使用ruby脚本搭建集群。需要ruby的运行环境。中间遇到确认[y/N]就选y

yum install ruby



yum install rubygems



将ruby脚本运行使用的包上传到/usr/loca/software下



执行安装：

gem install redis-3.2.2.gem



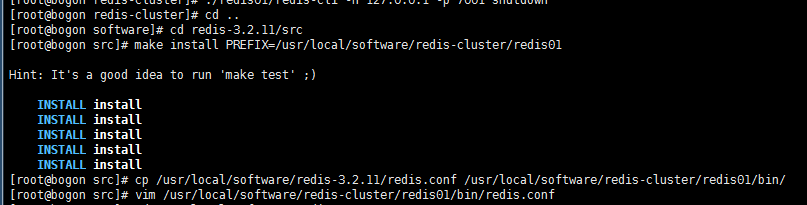
## 安装redis

使用集群版先要安装单机版redis的。注意安装集群的时候要求每个节点都是空的。所以不要复制以前的redis，最好重新安装一个。

创建集群目录redis-cluster



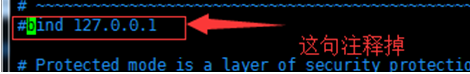
先安装第一个节点到redis-cluster/redis01下

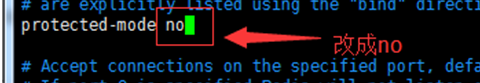


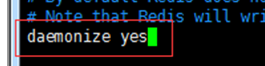
Redis集群中应该有奇数个节点，所以至少应该有三个节点，如果其中的一个节点挂掉，集群将不能访问。redis cluster 为了保证数据的高可用性，加入了主从模式，一个主节点对应一个或多个从节点，主节点提供数据存取，从节点则是从主节点拉取数据备份，当这个主节点挂掉后，就会有这个从节点选取一个来充当主节点，从而保证集群不会挂掉。如果一个主节点和它所有的从节点都挂掉了，那集群也就不能正常的服务了。

Redis集群至少需要6台服务器。搭建伪分布式。可以使用一台虚拟机运行6个redis实例。需要修改redis的端口号7001-7006

修改redis01下的redis.conf文件



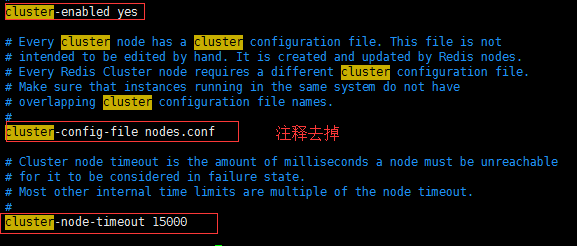


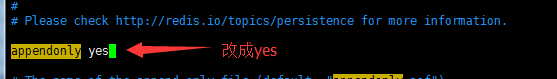


端口改成7001



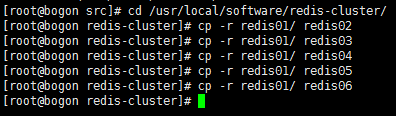
配置文件中还需要把cluster相关配置前的注释去掉





保存并退出

在redis-cluster目录下，将redis01复制5份



分别修改redis02-redis06的端口为7002-7006

启动每个redis。最好提前都ping一下每个节点。

可以在redis-cluster目录下创建启动集群的脚本：start-cluster.sh

cd redis01/bin

./redis-server redis.conf

cd ../../redis02/bin

./redis-server redis.conf

cd ../../redis03/bin

./redis-server redis.conf

cd ../../redis04/bin

./redis-server redis.conf

cd ../../redis05/bin

./redis-server redis.conf

cd ../../redis06/bin

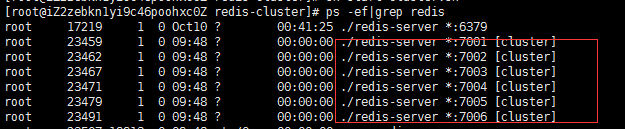
./redis-server redis.conf

cd ../../

授权chmod u+x start-cluster.sh

(u是自己，x是可执行)

查看运行状态：



## 创建集群

把创建集群的ruby脚本redis-trib.rb复制到redis-cluster文件夹下

cp /usr/local/software/redis-3.2.11/src/redis-trib.rb /usr/local/software/redis-cluster



注意安装集群的时候，防火墙中不仅要放开redis的端口，还要放开，而且需要开通集群总线端口

集群总线端口为redis客户端连接的端口 + 10000

如redis端口为6379

则集群总线端口为16379

所以当前的机器上要放开17001-17006端口，否则安装集群的时候会一直Waiting for the cluster to join

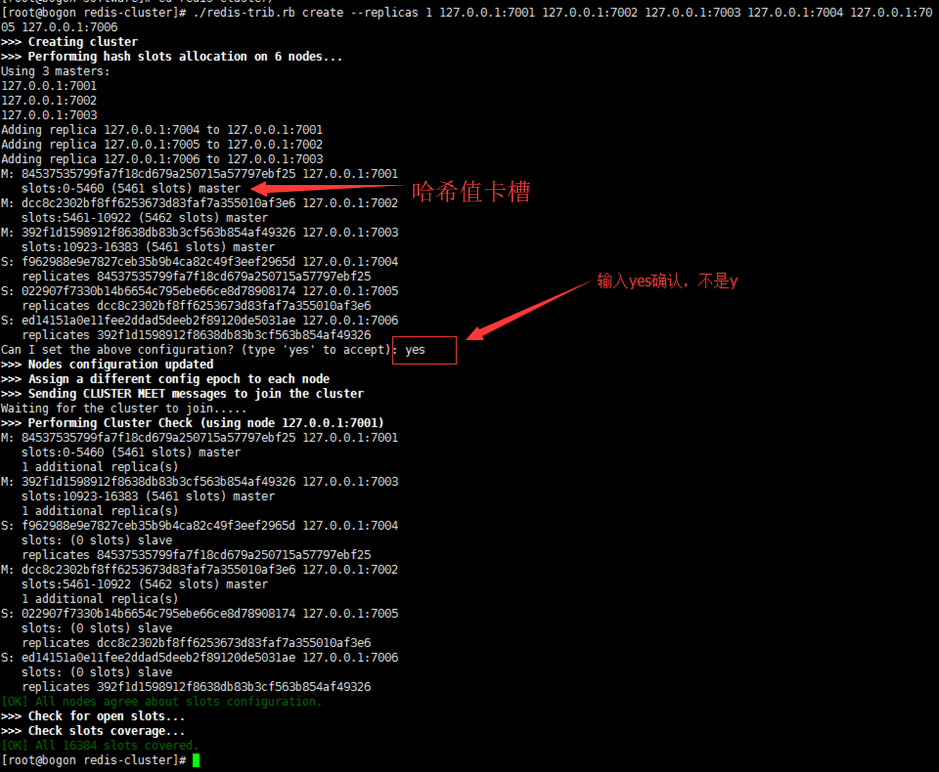
开放端口：

/sbin/iptables -I INPUT -p tcp --dport 6379 -j ACCEPT  
保存  
/etc/rc.d/init.d/iptables save  
重启服务  
service iptables restart

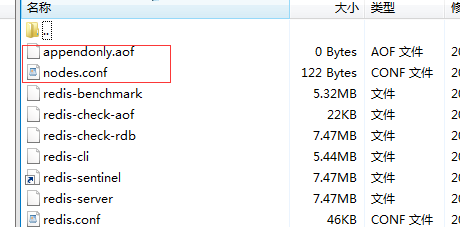
执行安装

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

其中主机ip不能使用内网ip或者127.0.0.1，否则远程无法连接集群。这里不支持域名。



如果集群创建失败了，需要把每个节点下的集群文件删除才能再次创建：

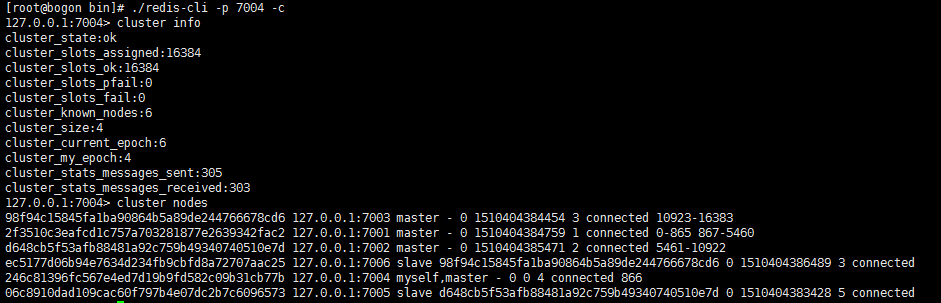


测试连接：

./redis-cli -p 7002 -c

其中-c代表集群

cluster info       打印集群的信息  
cluster nodes   列出集群当前已知的所有节点(node)，以及这些节点的相关信息



上面节点的ip是127.0.0.1，是在内网搭建的，外网无法访问该集群。自己搭建的时候需要换成外网ip。

创建停止集群的脚本stop-cluster.sh：

redis01/bin/redis-cli -p 7001 shutdown

redis01/bin/redis-cli -p 7002 shutdown

redis01/bin/redis-cli -p 7003 shutdown

redis01/bin/redis-cli -p 7004 shutdown

redis01/bin/redis-cli -p 7005 shutdown

redis01/bin/redis-cli -p 7006 shutdown

## 设置集群密码

在集群安装成功前不要设置密码，先完成3.2节中java连接测试，再回来设置密码。

编辑每一个节点的/bin/redis.conf，在文件末尾追加下面两条配置，密码设置成123456 ：  
masterauth 123456

requirepass 123456

重启每一个节点。

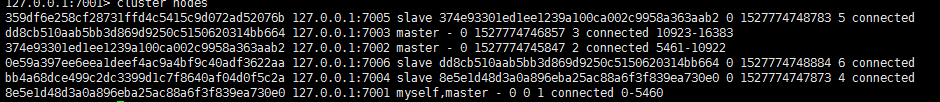
如果远程使用代码无法连接成功，把nodes.conf文件中myself节点的ip改成外网的，重新启动

连接命令：-a 后面是密码

./redis-cli -h 127.0.0.1 -p 7001 -c -a 123456

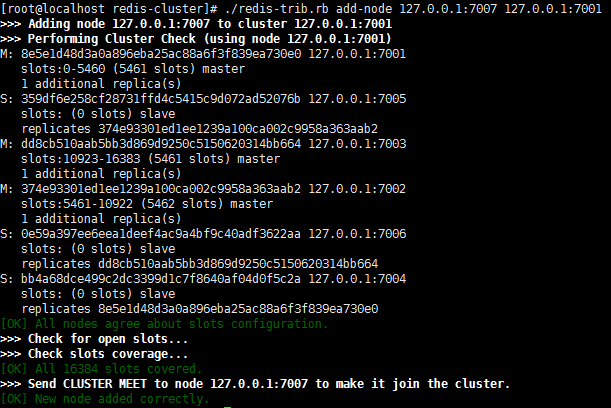
## 节点操作

### 3.5.1 追加节点



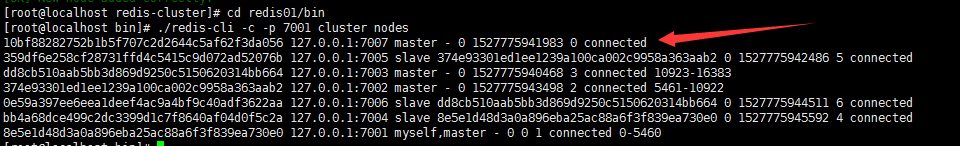
安装并启动7007节点的redis,注意不能设置密码，追加节点到集群中(任意一个集群节点都可以)

./redis-trib.rb add-node 127.0.0.1:7007 127.0.0.1:7001



查看新追加的节点。可以看到它还没有槽。

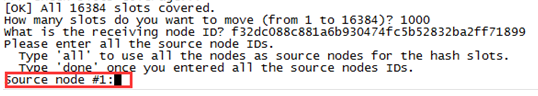
./redis-cli -c -p 7001 cluster nodes



移动一些槽到新节点上

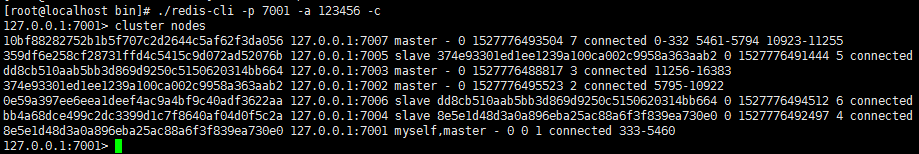
./redis-trib.rb reshard 127.0.0.1:7001

接着输入想要移动的槽数，如1000



接着输入all，表示所有master都移动，凑够1000个槽

连接集群，查看，可以看见新的节点被分配了槽：



把7008也添加到集群中

./redis-trib.rb add-node 127.0.0.1:7008 127.0.0.1:7001

把7008编程7007的从节点

./redis-cli -c -p 7008 cluster replicate 10bf88282752b1b5f707c2d2644c5af62f3da056

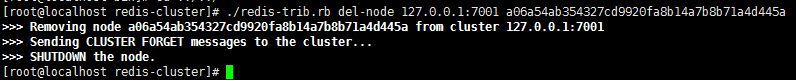


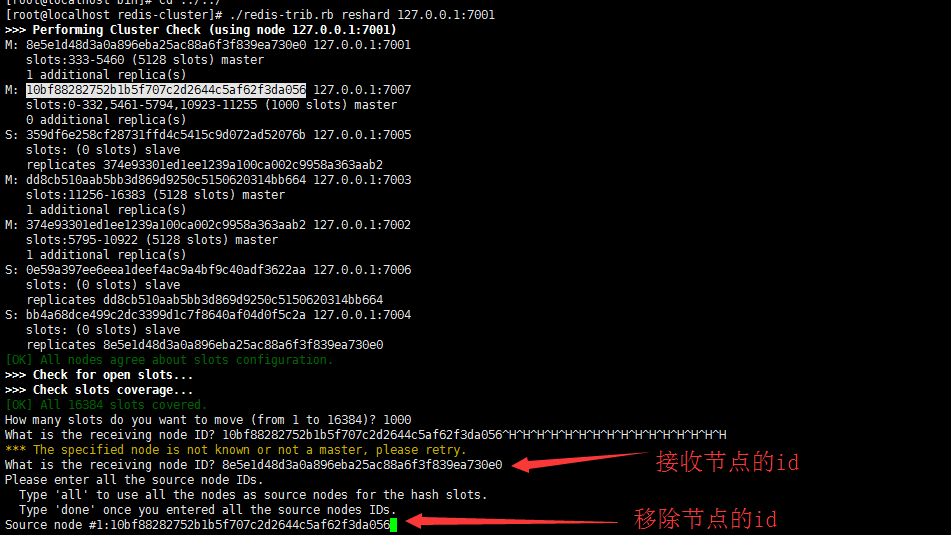
### 3.5.2 删除节点

删除从节点：

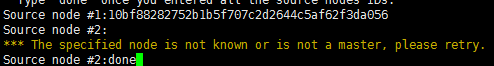
连接任意一个节点，最后一个参数是要删除的从节点id

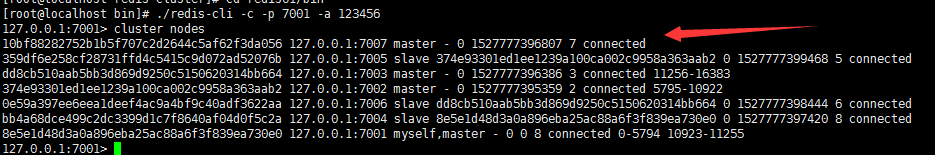
./redis-trib.rb del-node 127.0.0.1:7001 a06a54ab354327cd9920fa8b14a7b8b71a4d445a



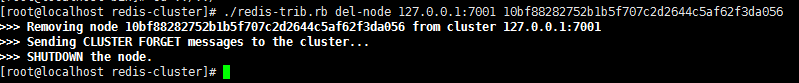


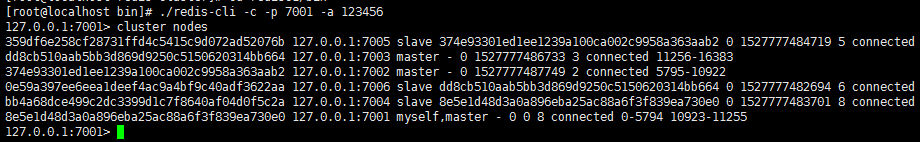
输入done





删除主节点





# JAVA操作redis

java操作redis需要引入jedis的jar包

<!--Jedis的jar-->  
<dependency>  
 <groupId>redis.clients</groupId>  
 <artifactId>jedis</artifactId>  
 <version>2.7.2</version>  
</dependency>

与spring结合还要额外引入spring的jar

<!--Spring基础jar包-->  
<dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-context</artifactId>  
 <version>4.3.11.RELEASE</version>  
</dependency>

## 单机版

### 使用Jedis

**@Test  
public void** testJedis() **throws Exception {  
 // 第一步：创建一个Jedis对象。需要指定服务端的ip及端口。  
 Jedis jedis = new** Jedis(**"www.vm.com"**, **6379**);  
 **System**.***out***.println(**jedis**.ping());  
 **// 第二步：使用Jedis对象操作数据库，每个redis命令对应一个方法。  
 jedis**.set(**"hello"**,**"123"**);  
 **System**.***out***.println(**jedis**.get(**"hello"**));  
 **// 第三步：关闭Jedis  
 jedis**.close();  
**}**

密码连接用：需要2.9.0的jar包

**jedis**.auth(**"123456"**);

### 使用Spring

在spring配置文件中配置redis连接池的信息：

<bean id="jedisPool" class="redis.clients.jedis.JedisPool">  
 <constructor-arg name="host" value="www.vm.com"></constructor-arg>  
 <constructor-arg name="port" value="6379"></constructor-arg>  
</bean>

测试方法：

**@Test  
public void** testSpringJedis() **throws Exception {** ApplicationContext **ac =  
 new** ClassPathXmlApplicationContext(**"spring-redis.xml"**);  
 **//获取连接池对象  
 JedisPool jedisPool =** (**JedisPool**) **ac**.**getBean**(**"jedisPool"**);  
 **//从连接池中获取连接  
 Jedis jedis = jedisPool**.getResource();  
 **System**.***out***.println(**jedis**.ping());  
 **System**.***out***.println(**jedis**.exists(**"hello"**));  
 **jedis**.close();**//关闭连接  
 jedisPool**.close();**//关闭连接池  
}**

如果使用密码：

<bean id="jedisPoolConfig" class="redis.clients.jedis.JedisPoolConfig">  
</bean>  
  
<bean id="jedisPool" class="redis.clients.jedis.JedisPool">  
 <constructor-arg name="host" value="www.xiaoheiben.com"></constructor-arg>  
 <constructor-arg name="port" value="6379"></constructor-arg>  
 <constructor-arg name="poolConfig" ref="jedisPoolConfig"></constructor-arg>  
 <constructor-arg name="password" value="123456"></constructor-arg>  
 <constructor-arg name="timeout" value="30000"></constructor-arg>  
</bean>

## 集群版

### 使用JedisCluster

**@Test  
public void** testJedisCluster() **throws Exception {  
 // 第一步：需要一个Set<HostAndPort>参数,存放Redis节点的列表。** Set**<HostAndPort> nodes = new** HashSet**<>**();  
 **nodes**.**add**(**new** HostAndPort(**"127.0.0.1"**, **7001**));  
 **nodes**.**add**(**new** HostAndPort(**"127.0.0.1"**, **7002**));  
 **nodes**.**add**(**new** HostAndPort(**"127.0.0.1"**, **7003**));  
 **nodes**.**add**(**new** HostAndPort(**"127.0.0.1"**, **7004**));  
 **nodes**.**add**(**new** HostAndPort(**"127.0.0.1"**, **7005**));  
 **nodes**.**add**(**new** HostAndPort(**"127.0.0.1"**, **7006**));  
 **JedisCluster jedisCluster = new** JedisCluster(**nodes**);  
 **// 第二步：直接使用JedisCluster对象操作redis。在系统中单例存在。  
 System**.***out***.println(**jedisCluster**.exists(**"hello"**));  
 **// 第三步：系统关闭前，关闭JedisCluster对象。  
 jedisCluster**.close();  
**}**

注意使用JedisCluster连接阿里云的redis集群时，Jedis的jar使用2.9.0版本可能连不上，换2.7.2即可。如果测试密码连接，可以使用2.9.0的jar，因为2.9.0才开始支持密码。

**JedisPoolConfig config = new** JedisPoolConfig();  
**JedisCluster jedisCluster = new** JedisCluster(**nodes**, **3000**, **1000**, **5**, **"123456"**, **config**);

### 使用Spring

<!-- 集群版的配置 -->  
<bean id="jedisCluster" class="redis.clients.jedis.JedisCluster">  
 <constructor-arg>  
 <set>  
 <bean class="redis.clients.jedis.HostAndPort">  
 <constructor-arg name="host" value="127.0.0.1"></constructor-arg>  
 <constructor-arg name="port" value="7001"></constructor-arg>  
 </bean>  
 <bean class="redis.clients.jedis.HostAndPort">  
 <constructor-arg name="host" value="127.0.0.1"></constructor-arg>  
 <constructor-arg name="port" value="7002"></constructor-arg>  
 </bean>  
 <bean class="redis.clients.jedis.HostAndPort">  
 <constructor-arg name="host" value="127.0.0.1"></constructor-arg>  
 <constructor-arg name="port" value="7003"></constructor-arg>  
 </bean>  
 <bean class="redis.clients.jedis.HostAndPort">  
 <constructor-arg name="host" value="127.0.0.1"></constructor-arg>  
 <constructor-arg name="port" value="7004"></constructor-arg>  
 </bean>  
 <bean class="redis.clients.jedis.HostAndPort">  
 <constructor-arg name="host" value="127.0.0.1"></constructor-arg>  
 <constructor-arg name="port" value="7005"></constructor-arg>  
 </bean>  
 <bean class="redis.clients.jedis.HostAndPort">  
 <constructor-arg name="host" value="127.0.0.1"></constructor-arg>  
 <constructor-arg name="port" value="7006"></constructor-arg>  
 </bean>  
 </set>  
 </constructor-arg>  
</bean>

测试密码连接：

<bean id="jedisPoolConfig" class="redis.clients.jedis.JedisPoolConfig">  
</bean>  
<bean id="jedisCluster" class="redis.clients.jedis.JedisCluster">  
 <constructor-arg name="jedisClusterNode">  
 <set>  
 <bean class="redis.clients.jedis.HostAndPort">  
 <constructor-arg name="host" value="10.0.107.156"></constructor-arg>  
 <constructor-arg name="port" value="6381"></constructor-arg>  
 </bean>  
 <bean class="redis.clients.jedis.HostAndPort">  
 <constructor-arg name="host" value="10.0.107.156"></constructor-arg>  
 <constructor-arg name="port" value="6382"></constructor-arg>  
 </bean>  
 <bean class="redis.clients.jedis.HostAndPort">  
 <constructor-arg name="host" value="10.0.107.156"></constructor-arg>  
 <constructor-arg name="port" value="6383"></constructor-arg>  
 </bean>  
 <bean class="redis.clients.jedis.HostAndPort">  
 <constructor-arg name="host" value="10.0.107.156"></constructor-arg>  
 <constructor-arg name="port" value="6384"></constructor-arg>  
 </bean>  
 <bean class="redis.clients.jedis.HostAndPort">  
 <constructor-arg name="host" value="10.0.107.156"></constructor-arg>  
 <constructor-arg name="port" value="6385"></constructor-arg>  
 </bean>  
 <bean class="redis.clients.jedis.HostAndPort">  
 <constructor-arg name="host" value="10.0.107.156"></constructor-arg>  
 <constructor-arg name="port" value="6386"></constructor-arg>  
 </bean>  
 </set>  
 </constructor-arg>  
 <constructor-arg name="password" value="123456"/>  
 <constructor-arg name="connectionTimeout" value="3000"/>  
 <constructor-arg name="soTimeout" value="1000"/>  
 <constructor-arg name="maxAttempts" value="5"/>  
 <constructor-arg name="poolConfig" ref="jedisPoolConfig"/>  
</bean>

测试代码：

**@Test  
public void** testSpringJedisCluster()**{** ApplicationContext **ac =  
 new** ClassPathXmlApplicationContext(**"spring-redis.xml"**);  
 **JedisCluster jedisCluster =** (**JedisCluster**)**ac**.**getBean**(**"jedisCluster"**);  
 **System**.***out***.println(**jedisCluster**.exists(**"hello"**));  
 **jedisCluster**.close();  
**}**

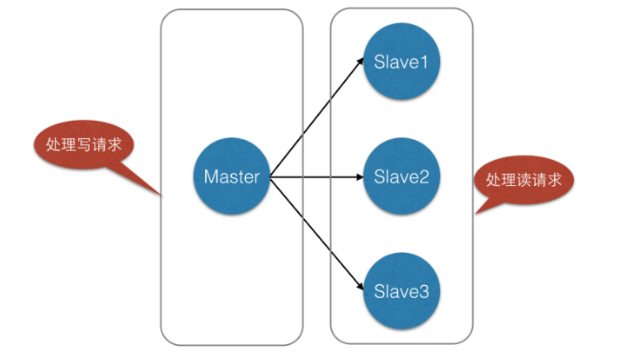
# Redis主从和哨兵

## 主从复制

一般来说，要将Redis运用于工程项目中，只使用一台Redis是万万不能的，原因如下：

从结构上，单个Redis服务器会发生单点故障，并且一台服务器需要处理所有的请求负载，压力较大；

从容量上，单个Redis服务器内存容量有限，就算一台Redis服务器内容容量为256G，也不能将所有内容用作Redis存储内存，一般来说，单台Redis最大使用内存不应该超过20G。



如图所示，将一台Redis服务器作主库(Matser)，其他三台作为从库(Slave)，主库只负责写数据，每次有数据更新都将更新的数据同步到它所有的从库，而从库只负责读数据。

这样一来，就有了两个好处：

读写分离，不仅可以提高服务器的负载能力并且可以根据读请求的规模自由增加或者减少从库的数量棒极了；

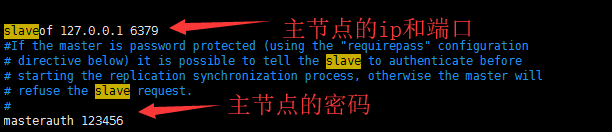
数据被复制成了了好几份，就算有一台机器出现故障，也可以使用其他机器的数据快速恢复。

需要注意的是：在Redis主从模式中，一台主库可以拥有多个从库，但是一个从库只能隶属于一个主库。

实现主从复制（Master-Slave Replication）的工作原理：Slave从节点服务启动并连接到Master之后，它将主动发送一个SYNC命令。Master服务主节点收到同步命令后将启动后台存盘进程，同时收集所有接收到的用于修改数据集的命令，在后台进程执行完毕后，Master将传送整个数据库文件到Slave，以完成一次完全同步。而Slave从节点服务在接收到数据库文件数据之后将其存盘并加载到内存中。此后，Master主节点继续将所有已经收集到的修改命令，和新的修改命令依次传送给Slaves，Slave将在本次执行这些数据修改命令，从而达到最终的数据同步。  
        如果Master和Slave之间的链接出现断连现象，Slave可以自动重连Master，但是在连接成功之后，一次完全同步将被自动执行。

做主从复制的时候，主节点不需要做任何变动。从节点需要做如下配置：

(同一台机器上模拟，记得改端口)



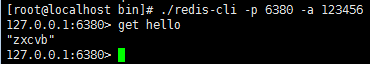
启动从节点

在bin目录下执行./redis-server redis.conf

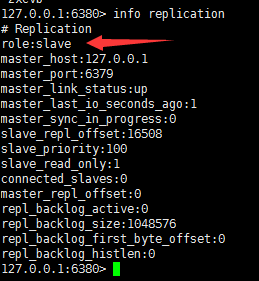
在主节点上添加数据：



去从节点查询数据：



使用info replication查看节点情况。



从节点是只读节点，不能添加数据。如果主节点挂了，那么就不能再添加数据了。



## 哨兵模式

Sentinel(哨兵)进程是用于监控redis集群中Master主服务器工作的状态，在Master主服务器发生故障的时候，可以实现Master和Slave服务器的切换，保证系统的高可用，其已经被集成在redis2.6+的版本中，Redis的哨兵模式到了2.8版本之后就稳定了下来。一般在生产环境也建议使用Redis的2.8版本的以后版本。哨兵(Sentinel) 是一个分布式系统，你可以在一个架构中运行多个哨兵(sentinel) 进程，这些进程使用流言协议(gossipprotocols)来接收关于Master主服务器是否下线的信息，并使用投票协议(Agreement Protocols)来决定是否执行自动故障迁移,以及选择哪个Slave作为新的Master。每个哨兵(Sentinel)进程会向其它哨兵(Sentinel)、Master、Slave定时发送消息，以确认对方是否”活”着，如果发现对方在指定配置时间(可配置的)内未得到回应，则暂时认为对方已掉线，也就是所谓的”主观认为宕机” ，英文名称：Subjective Down，简称SDOWN。有主观宕机，肯定就有客观宕机。当“哨兵群”中的多数Sentinel进程在对Master主服务器做出 SDOWN 的判断，并且通过 SENTINEL is-master-down-by-addr 命令互相交流之后，得出的Master Server下线判断，这种方式就是“客观宕机”，英文名称是：Objectively Down， 简称 ODOWN。通过一定的vote算法，从剩下的slave从服务器节点中，选一台提升为Master服务器节点，然后自动修改相关配置，并开启故障转移（failover）。

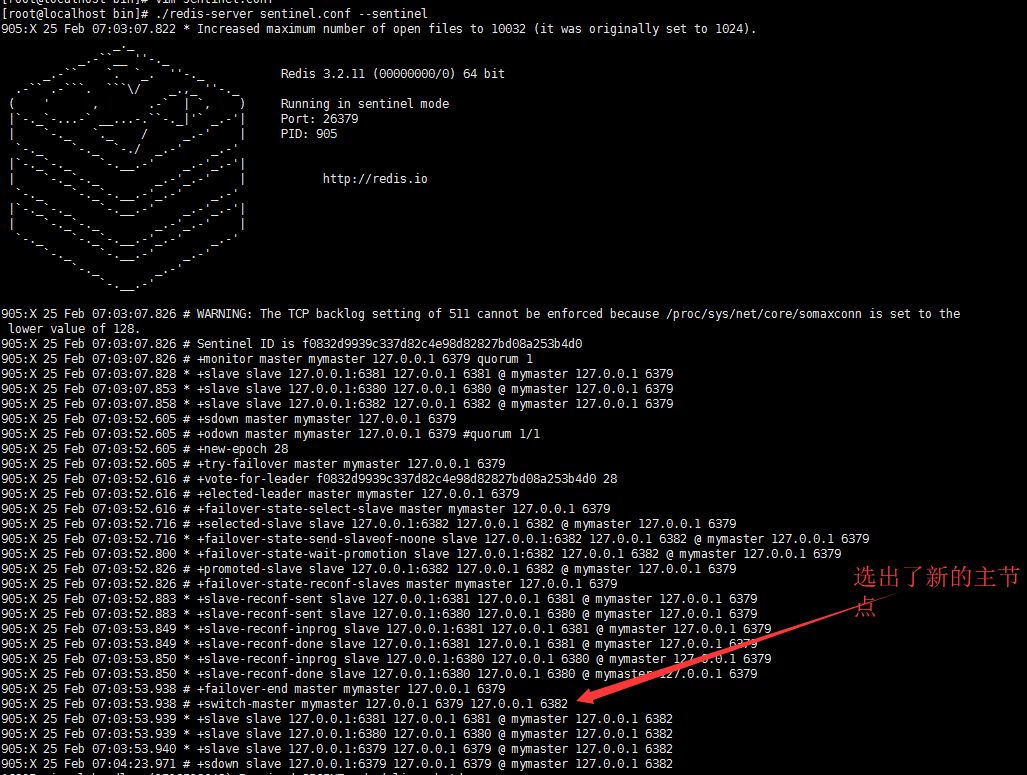
复制安装包中/usr/local/software/redis-3.2.11/sentinel.conf 到redis的bin目录

修改sentinel.conf



以哨兵模式开启

./redis-server sentinel.conf --sentinel



关闭主节点，会看到哨兵重新选择了主节点

哨兵支持集群

注意：故障切换的两个条件：

1.quorum个哨兵发现master不可用

2.半数以上的哨兵可用

1)：每个Sentinel以每秒钟一次的频率向它所知的Master，Slave以及其他 Sentinel 实例发送一个 PING 命令   
2)：如果一个实例（instance）距离最后一次有效回复 PING 命令的时间超过 down-after-milliseconds 选项所指定的值， 则这个实例会被 Sentinel 标记为主观下线。   
3)：如果一个Master被标记为主观下线，则正在监视这个Master的所有 Sentinel 要以每秒一次的频率确认Master的确进入了主观下线状态。   
4)：当有足够数量的 Sentinel（大于等于配置文件指定的值）在指定的时间范围内确认Master的确进入了主观下线状态， 则Master会被标记为客观下线   
5)：在一般情况下， 每个 Sentinel 会以每 10 秒一次的频率向它已知的所有Master，Slave发送 INFO 命令   
6)：当Master被 Sentinel 标记为客观下线时，Sentinel 向下线的 Master 的所有 Slave 发送 INFO 命令的频率会从 10 秒一次改为每秒一次   
7)：若没有足够数量的 Sentinel 同意 Master 已经下线， Master 的客观下线状态就会被移除。

8)：若 Master 重新向 Sentinel 的 PING 命令返回有效回复， Master 的主观下线状态就会被移除。

下面我们来解释一下两个“下线”的概念，一个是“主观下线”，另一个就是“客观下线”。

**主观下线（Subjectively Down， 简称 SDOWN）指的是单个 Sentinel 实例对服务器做出的下线判断。**

**客观下线（Objectively Down， 简称 ODOWN）指的是多个 Sentinel 实例在对同一个服务器做出 SDOWN 判断，并且通过 SENTINEL is-master-down-by-addr 命令互相交流之后，得出的服务器下线判断。（一个 Sentinel 可以通过向另一个 Sentinel 发送 SENTINEL is-master-down-by-addr 命令来询问对方是否认为给定的服务器已下线。）**

如果一个服务器没有在 master-down-after-milliseconds 选项所指定的时间内，对向它发送 PING 命令的 Sentinel（哨兵）进程返回一个有效回复（valid reply），那么Sentinel（哨兵）进程就会将这个服务器标记为主观下线。

服务器对 PING 命令的有效回复可以是以下三种回复的其中一种：

1、返回 +PONG 。

2、返回 -LOADING错误。

3、返回 -MASTERDOWN错误。

如果服务器返回除以上三种回复之外的其他回复，又或者在指定时间内没有回复 PING 命令，那么 Sentinel（哨兵）进程认为服务器返回的回复无效（non-valid）。

如果一个服务器在 master-down-after-milliseconds 毫秒内，一直返回无效回复才会被 Sentinel 标记为主观下线。

举个例子，如果 master-down-after-milliseconds 选项的值为 30000 毫秒（30 秒），那么只要服务器能在每 29 秒之内返回至少一次有效回复， 这个服务器就仍然会被认为是处于正常状态的。

**从“主观下线”状态切换到“客观下线”状态并没有使用严格的法定人数算法（strong quorum algorithm），而是使用了流言协议，该协议解释为：如果 Sentinel（哨兵）进程在给定的时间范围内，从其他 Sentinel（哨兵）进程那里接收到了足够数量的主服务器下线报告， 那么 Sentinel（哨兵）进程就会将主服务器的状态从“主观下线”改变为“客观下线”。如果之后其他 Sentinel（哨兵）进程不再报告主服务器已下线，那么“客观下线”状态就会被移除。**

**“客观下线”条件只适用于主服务器：对于任何其他类型的 Redis 实例， Sentinel（哨兵）进程在将它们判断为下线前不需要进行协商，所以Slave从服务器或者其他 Sentinel（哨兵）进程永远不会达到“客观下线”条件。**

**只要有一个 Sentinel（哨兵）进程发现某个主服务器进入了“客观下线”状态，这个 Sentinel（哨兵）进程就可能会被其他 Sentinel（哨兵）进程推选出，并对失效的主服务器执行自动故障迁移操作。**

**Sentinel（哨兵）配置文件简介**

在Redis的源码中包含了一个名为 sentinel.conf 的文件， 这个文件就是带有注释的Sentinel（哨兵）的配置文件的示例。

如果想要运行一个“哨兵”程序，以下配置项是最少配置：  
**sentinel monitor mymaster 127.0.0.1 6379 1**

**sentinel down-after-milliseconds mymaster 60000**

**sentinel failover-timeout mymaster 180000**

**sentinel parallel-syncs mymaster 1**

第一行配置表示 Sentinel（哨兵）进程去监视一个名为 mymaster 的主服务器，这个主服务器的 IP 地址为 127.0.0.1 ， 端口号为 6379，而将这个主服务器判断为失效至少需要 1 个 Sentinel（哨兵）进程的同意。如果在架构系统中已经配置类多个Sentinel（哨兵）进程，在同意“Master主服务器”下线的 Sentinel（哨兵）进程的数量不达标的情况下，Sentinel（哨兵）进程就不会执行自动故障迁移。在设置多Sentinel（哨兵）进程的情况下，无论设置多少个 Sentinel（哨兵）进程同意才能判断一个服务器失效，一个 Sentinel 都需要获得架构系统中多数 Sentinel（哨兵）进程的支持， 才能发起一次自动故障迁移，并预留一个给定的配置纪元 （configuration Epoch ，一个配置纪元就是一个新主服务器配置的版本号）。如果您只配置了一个Sentinel（哨兵）进程来做监控，那一个Sentinel（哨兵）进程也可以决定“Master主服务器”是否下线。

**其他选项的基本格式如下：sentinel <选项的名字> <主服务器的名字> <选项的值>**

配置选项的解释如下：

**1、down-after-milliseconds ： Sentinel（哨兵）进程判断服务器已经掉线所需的毫秒数。**

如果被监控的服务器在给定的毫秒数之内，并没有返回 Sentinel（哨兵）进程发送的 PING 命令的回复，或者返回一个错误，那么 Sentinel（哨兵）进程将这个服务器标记为主观下线（subjectively down，简称 SDOWN ）。如果在架构系统中配置了多个Sentinel（哨兵）进程的情况下，只有一个Sentinel（哨兵）进程将服务器标记为主观下线并不一定会引起服务器的自动故障迁移，只有在足够数量的 Sentinel（哨兵）进程都将一个服务器标记为主观下线之后，服务器才会被标记为客观下线（objectively down， 简称 ODOWN ），这时才回执行自动故障迁移。另外一种情况是，在架构系统中只配置了一个Sentinel（哨兵）进程的话，那这Sentinel（哨兵）进程也可以决定被监控的服务器的是否“下线”。

将服务器标记为客观下线所需的 Sentinel（哨兵）进程数量由对主服务器的配置决定。

**2、parallel-syncs ：在执行故障转移时，最多可以有多少个从服务器同时对新的主服务器进行同步，这个数字越小，完成故障转移所需的时间就越长。**

如果“Slave从服务器”被设置为允许使用过期数据集（参见对 redis.conf 文件中对 slave-serve-stale-data 选项的说明），那么你可能不希望所有“Slave从服务器”都在同一时间向新的“Master主服务器”发送同步请求， 因为尽管复制过程的绝大部分步骤都不会阻塞“Slave从服务器”，但“Slave从服务器”在载入“Master主服务器”发来的 RDB 文件时， 仍然会造成“Slave从服务器”在一段时间内不能处理命令请求，如果全部“Slave从服务器”一起对新的“Master主服务器”进行同步， 那么就可能会造成所有“Slave从服务器”在短时间内全部不可用的情况出现。你可以通过将这个值设为 1 来保证每次只有一个Slave从服务器处于不能处理命令请求的状态。

**3、failover-timeout：实现主从切换，完成故障转移的所需要的最大时间值。若Sentinel（哨兵）进程在该配置值内未能完成故障转移的操作（即故障时master/slave自动切换），则认为本次故障转移操作失败。**

**4、notification-script: 指定Sentinel（哨兵）进程检测到Master-Name所指定的“Master主服务器”的实例异常的时候，所要调用的报警脚本。该配置项可选，但线上系统建议配置。**

**哨兵模式的优缺点**

**优点：**

1、哨兵集群模式是基于主从模式的，所有主从的优点，哨兵模式同样具有。

2、主从可以切换，故障可以转移，系统可用性更好。

3、哨兵模式是主从模式的升级，系统更健壮，可用性更高。

**缺点：**

1、Redis较难支持在线扩容，在集群容量达到上限时在线扩容会变得很复杂。为避免这一问题，运维人员在系统上线时必须确保有足够的空间，这对资源造成了很大的浪费。