

Redis

整理自互联网，仅供学习交流。



2018-10-30

[公司名称]

[公司地址]

目录

# Redis

## 入门

### NoSQL介绍

为什么有NOSql:

互联网项目:特点: 数据库高并发读写,海量数据高效率存储,可扩展

NOSQL主流:

键值对(redis),列式存储,文档类型,图形数据库

NoSql特点:

特点: 数据库高并发读写,海量数据高效率存储,可扩展

NoSQL 是Not Only SQL 的缩写，意即"不仅仅是SQL"的意思，泛指非关系型的数据库。强调Key-Value Stores和文档数据库的优点，而不是单纯的反对RDBMS。

NoSQL产品是传统关系型数据库的功能阉割版本，通过减少用不到或很少用的功能，来大幅度提高产品性能

NoSQL产品 redis、mongodb [Membase](http://www.couchbase.org/membase)、[HBase](http://hbase.apache.org/)

### Redis 简介

## Redis简介

什么是实时系统: 瞬时可以实现某些数据的统计或者是功能的实现

\*\_Redis由来:

\*\_什么是redis?

由C语言实现的直接操作内存的开源的高性能的数据库软件

\*\_redis应用场景

缓存（数据查询，短连接，新闻内容、商品内容等等）最多使用

聊天室的在线好友列表

任务队列（秒杀、抢购、12306）

应用排行榜

网站访问统计

#### 什么是Redis?

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：

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。

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。

Redis支持数据的备份，即master-slave模式的数据备份。

#### Redis应用场景

主要能够体现 解决数据库的访问压力。

例如:短信验证码时间有效期、session共享解决方案

#### Redis优势

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

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。

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

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？

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。

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的安装

##### Redis的安装

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++

安装步骤：

第一步：redis的源码包上传到linux系统。

第二步：解压缩redis。

第三步：编译。进入redis源码目录。make

第四步：安装。make install PREFIX=/usr/local/redis

PREFIX参数指定redis的安装目录。一般软件安装到/usr目录下

##### 安装redis版本2

redis安装包: 源码形式提供 .c文件

将.c编译为.o文件 需要安装: gcc

将编译后的文件安装在linux系统上

等价于 希望在当前的系统安装JAVA版本的QQ,下载到的安装文件以.java源码形式提供.

需要:1\_下载GCC 在系统上有JDK(GCC),

2\_编译: 通过JDK将源码编译为字节码

3\_安装: 将字节码进行打包设置 .bat

1\_获取安装包 redis-3.0.7.tar.gz

2\_利用filezilla.exe 将安装包上传到/root/

3\_cd /root/ \_\_\_> tar -zxvf redis-3.0.7.tar.gz

4\_编译 将源码翻译为.o文件

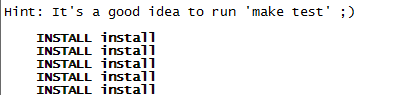
cd /root/redis-3.0.7 \_\_\_> make

5\_创建一个目录 /usr/loca/redis (安装目录)

6\_安装reids

cd /root/redis-3.0.7

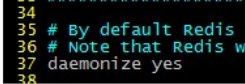
make PREFIX=/usr/local/redis install



7\_ 由于redis启动需要一个配置文件,将配置文件复制到 /root/redis/

cp /root/redis-3.0.7/redis.conf /root/redis

8\_ 修改/usr/local/redis/redis.conf



9\_ 启动服务端 ./bin/redis-server ./redis.conf

防火墙打开6379端口

/sbin/iptables -I INPUT -p tcp --dport 6379 -j ACCEPT

将打开的端口的访问规则保存在文件中

/etc/rc.d/init.d/iptables save

查看机器开放的端口号

/etc/init.d/iptables status

10\_启动客户端 ./bin/redis-cli

11\_测试redis是否正确安装成功

向服务器发送 ping 命令,服务器返回 pong

向服务器发送 set name tom

get name 命令可以看到可以获取到tom的信息

向服务器发送keys \*

可以查看服务器中一共有多少键值对的数据,可以查看到各个键值类型数据

##### 连接redis

###### redis的启动：

前端启动：在redis的安装目录下直接启动redis-server

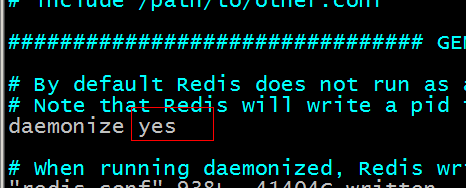
[root@localhost bin]# ./redis-server

后台启动：

把/root/redis-3.0.0/redis.conf复制到/usr/local/redis/bin目录下

[root@localhost redis-3.0.0]# cp redis.conf /usr/local/redis/bin/

修改配置文件：



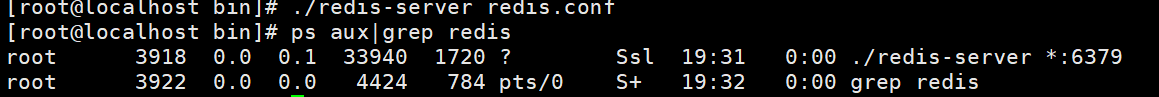
[root@localhost bin]# ./redis-server redis.conf

查看redis进程：

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

root 5190 0.1 0.3 33936 1712 ? Ssl 18:23 0:00 ./redis-server \*:6379

root 5196 0.0 0.1 4356 728 pts/0 S+ 18:24 0:00 grep redis

[root@localhost bin]# 

###### Redis-cli

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

默认连接localhost运行在6379端口的redis服务。

[root@localhost bin]# ./redis-cli -h 192.168.25.128 -p 6379

-h：连接的服务器的地址

-p：服务的端口号

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

#### Redis五种数据类型

String：key-value（做缓存）

Redis中所有的数据都是字符串。命令不区分大小写，key是区分大小写的。Redis是单线程的。Redis中不适合保存内容大的数据。

get、set、

incr：加一（生成id）

Decr：减一

Hash：key-fields-values（做缓存）

相当于一个key对于一个map，map中还有key-value

使用hash对key进行归类。

Hset：向hash中添加内容

Hget：从hash中取内容

List：有顺序可重复

192.168.25.128:6379> lpush list1 a b c d

(integer) 4

192.168.25.128:6379> lrange list1 0 -1

1) "d"

2) "c"

3) "b"

4) "a"

192.168.25.128:6379> rpush list1 1 2 3 4

(integer) 8

192.168.25.128:6379> lrange list1 0 -1

1) "d"

2) "c"

3) "b"

4) "a"

5) "1"

6) "2"

7) "3"

8) "4"

192.168.25.128:6379>

192.168.25.128:6379> lpop list1

"d"

192.168.25.128:6379> lrange list1 0 -1

1) "c"

2) "b"

3) "a"

4) "1"

5) "2"

6) "3"

7) "4"

192.168.25.128:6379> rpop list1

"4"

192.168.25.128:6379> lrange list1 0 -1

1) "c"

2) "b"

3) "a"

4) "1"

5) "2"

6) "3"

192.168.25.128:6379>

Set：元素无顺序，不能重复

192.168.25.128:6379> sadd set1 a b c c c d

(integer) 4

192.168.25.128:6379> smembers set1

1) "b"

2) "c"

3) "d"

4) "a"

192.168.25.128:6379> srem set1 a

(integer) 1

192.168.25.128:6379> smembers set1

1) "b"

2) "c"

3) "d"

192.168.25.128:6379>

还有集合运算命令，自学。

SortedSet（zset）：有顺序，不能重复

192.168.25.128:6379> zadd zset1 2 a 5 b 1 c 6 d

(integer) 4

192.168.25.128:6379> zrange zset1 0 -1

1) "c"

2) "a"

3) "b"

4) "d"

192.168.25.128:6379> zrem zset1 a

(integer) 1

192.168.25.128:6379> zrange zset1 0 -1

1) "c"

2) "b"

3) "d"

192.168.25.128:6379> zrevrange zset1 0 -1

1) "d"

2) "b"

3) "c"

192.168.25.128:6379> zrange zset1 0 -1 withscores

1) "c"

2) "1"

3) "b"

4) "5"

5) "d"

6) "6"

192.168.25.128:6379> zrevrange zset1 0 -1 withscores

1) "d"

2) "6"

3) "b"

4) "5"

5) "c"

6) "1"

192.168.25.128:6379>

##### 列出所有的keys

可以使用KEYS 命令

KEYS pattern

例如，

列出所有的key

redis> keys \*

列出匹配的key

redis>keys apple\*

1) apple1

2) apple2

##### Key命令

设置key的过期时间。

Expire key second：设置key的过期时间

Ttl key：查看key的有效期

Persist key：清除key的过期时间。Key持久化。

192.168.25.128:6379> expire Hello 100

(integer) 1

192.168.25.128:6379> ttl Hello

(integer) 77

#### Redis的持久化方案

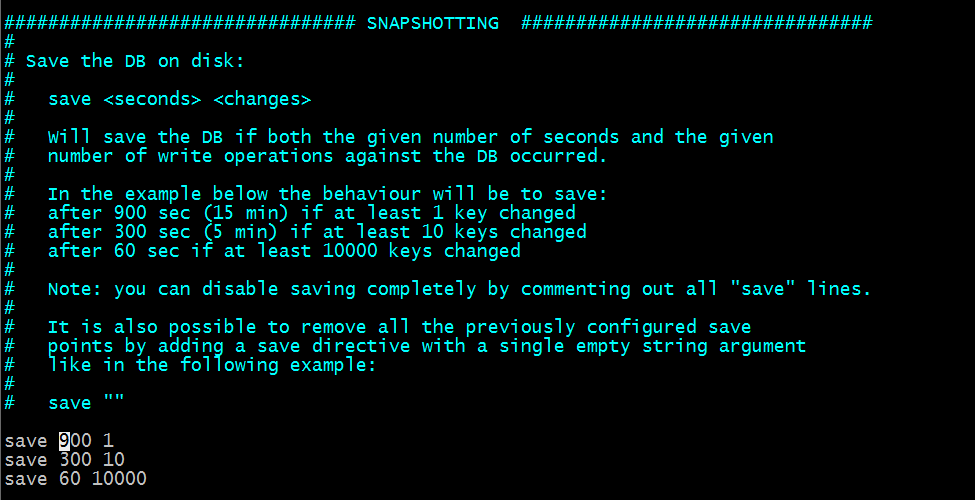
Redis的所有数据都是保存到内存中的。

Rdb：快照形式，定期把内存中当前时刻的数据保存到磁盘。Redis默认支持的持久化方案。

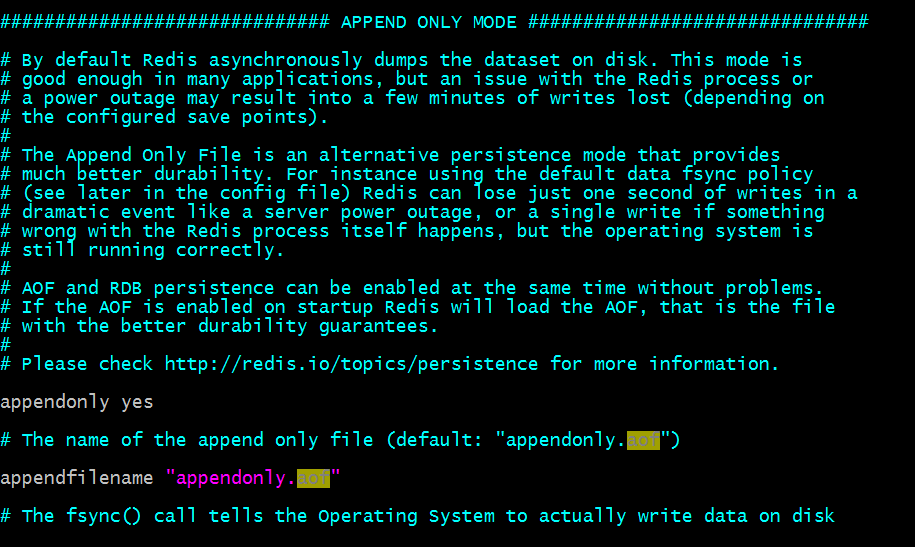
aof形式：append only file。把所有对redis数据库操作的命令，增删改操作的命令。保存到文件中。数据库恢复时把所有的命令执行一遍即可。

在redis.conf配置文件中配置。

Rdb：



Aof的配置：



两种持久化方案同时开启使用aof文件来恢复数据库。

#### Redis集群的搭建

##### redis-cluster架构图



redis-cluster投票:容错



架构细节:

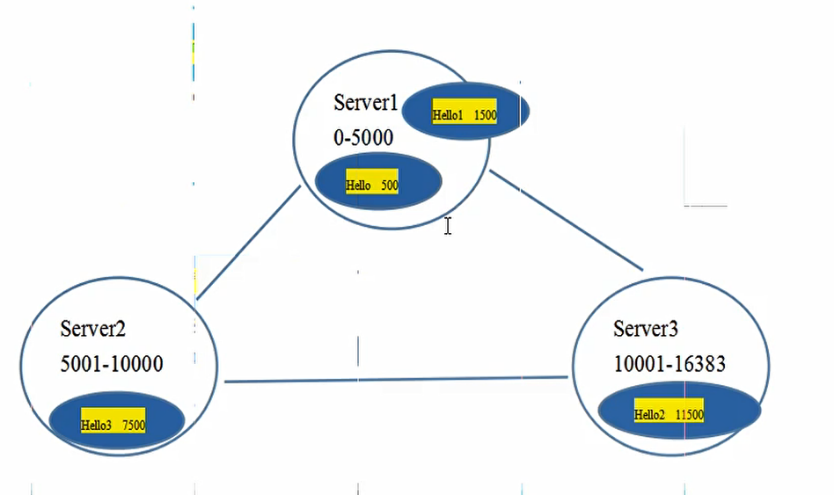
(1)所有的redis节点彼此互联(PING-PONG机制),内部使用二进制协议优化传输速度和带宽.

(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



##### Redis集群的搭建

Redis集群中至少应该有三个节点。要保证集群的高可用，需要每个节点有一个备份机。

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

###### 集群搭建环境

1、使用ruby脚本搭建集群。需要ruby的运行环境。

安装ruby

yum install ruby

yum install rubygems

1. 安装ruby脚本运行使用的包。

[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...

[root@localhost ~]#

[root@localhost ~]# cd redis-3.0.0/src

[root@localhost src]# ll \*.rb

-rwxrwxr-x. 1 root root 48141 Apr 1 2015 redis-trib.rb

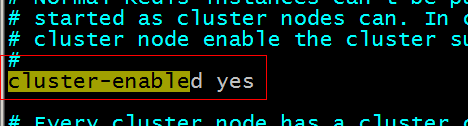
###### 搭建步骤

需要6台redis服务器。搭建伪分布式。

需要6个redis实例。

需要运行在不同的端口7001-7006

第一步：创建6个redis实例，每个实例运行在不同的端口。需要修改redis.conf配置文件。配置文件中还需要把cluster-enabled yes前的注释去掉。



第二步：启动每个redis实例。

第三步：使用ruby脚本搭建集群。

|  |
| --- |
| ./redis-trib.rb create --replicas 1 192.168.25.130:7001 192.168.25.130:7002 192.168.25.130:7003 192.168.25.130:7004 192.168.25.130:7005 192.168.25.130:7006 |

创建关闭集群的脚本：

[root@localhost redis-cluster]# vim shutdow-all.sh

redis01/redis-cli -p 7001 shutdown

redis01/redis-cli -p 7002 shutdown

redis01/redis-cli -p 7003 shutdown

redis01/redis-cli -p 7004 shutdown

redis01/redis-cli -p 7005 shutdown

redis01/redis-cli -p 7006 shutdown

[root@localhost redis-cluster]# chmod u+x shutdow-all.sh

|  |
| --- |
| [root@localhost redis-cluster]# ./redis-trib.rb create --replicas 1 192.168.25.128:7001 192.168.25.128:7002 192.168.25.128:7003 192.168.25.128:7004 192.168.25.128:7005 192.168.25.128:7006  >>> Creating cluster  Connecting to node 192.168.25.128:7001: OK  Connecting to node 192.168.25.128:7002: OK  Connecting to node 192.168.25.128:7003: OK  Connecting to node 192.168.25.128:7004: OK  Connecting to node 192.168.25.128:7005: OK  Connecting to node 192.168.25.128:7006: OK  >>> Performing hash slots allocation on 6 nodes...  Using 3 masters:  192.168.25.128:7001  192.168.25.128:7002  192.168.25.128:7003  Adding replica 192.168.25.128:7004 to 192.168.25.128:7001  Adding replica 192.168.25.128:7005 to 192.168.25.128:7002  Adding replica 192.168.25.128:7006 to 192.168.25.128:7003  M: 2e48ae301e9c32b04a7d4d92e15e98e78de8c1f3 192.168.25.128:7001  slots:0-5460 (5461 slots) master  M: 8cd93a9a943b4ef851af6a03edd699a6061ace01 192.168.25.128:7002  slots:5461-10922 (5462 slots) master  M: 2935007902d83f20b1253d7f43dae32aab9744e6 192.168.25.128:7003  slots:10923-16383 (5461 slots) master  S: 74f9d9706f848471583929fc8bbde3c8e99e211b 192.168.25.128:7004  replicates 2e48ae301e9c32b04a7d4d92e15e98e78de8c1f3  S: 42cc9e25ebb19dda92591364c1df4b3a518b795b 192.168.25.128:7005  replicates 8cd93a9a943b4ef851af6a03edd699a6061ace01  S: 8b1b11d509d29659c2831e7a9f6469c060dfcd39 192.168.25.128:7006  replicates 2935007902d83f20b1253d7f43dae32aab9744e6  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 192.168.25.128:7001)  M: 2e48ae301e9c32b04a7d4d92e15e98e78de8c1f3 192.168.25.128:7001  slots:0-5460 (5461 slots) master  M: 8cd93a9a943b4ef851af6a03edd699a6061ace01 192.168.25.128:7002  slots:5461-10922 (5462 slots) master  M: 2935007902d83f20b1253d7f43dae32aab9744e6 192.168.25.128:7003  slots:10923-16383 (5461 slots) master  M: 74f9d9706f848471583929fc8bbde3c8e99e211b 192.168.25.128:7004  slots: (0 slots) master  replicates 2e48ae301e9c32b04a7d4d92e15e98e78de8c1f3  M: 42cc9e25ebb19dda92591364c1df4b3a518b795b 192.168.25.128:7005  slots: (0 slots) master  replicates 8cd93a9a943b4ef851af6a03edd699a6061ace01  M: 8b1b11d509d29659c2831e7a9f6469c060dfcd39 192.168.25.128:7006  slots: (0 slots) master  replicates 2935007902d83f20b1253d7f43dae32aab9744e6  [OK] All nodes agree about slots configuration.  >>> Check for open slots...  >>> Check slots coverage...  [OK] All 16384 slots covered.  [root@localhost redis-cluster]# |

##### 集群的使用方法

Redis-cli连接集群。

[root@localhost redis-cluster]# redis01/redis-cli -p 7002 -c

-c：代表连接的是redis集群

#### Jedis

需要把jedis依赖的jar包添加到工程中。Maven工程中需要把jedis的坐标添加到依赖。

推荐添加到服务层。E3-content-Service工程中。

##### 连接单机版

第一步：创建一个Jedis对象。需要指定服务端的ip及端口。

第二步：使用Jedis对象操作数据库，每个redis命令对应一个方法。

第三步：打印结果。

第四步：关闭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("192.168.25.128", 6379);  // 第二步：使用Jedis对象操作数据库，每个redis命令对应一个方法。  String result = jedis.get("hello");  // 第三步：打印结果。  System.***out***.println(result);  // 第四步：关闭Jedis  jedis.close();  } |

##### 连接单机版使用连接池

第一步：创建一个JedisPool对象。需要指定服务端的ip及端口。

第二步：从JedisPool中获得Jedis对象。

第三步：使用Jedis操作redis服务器。

第四步：操作完毕后关闭jedis对象，连接池回收资源。

第五步：关闭JedisPool对象。

|  |
| --- |
| @Test  **public** **void** testJedisPool() **throws** Exception {  // 第一步：创建一个JedisPool对象。需要指定服务端的ip及端口。  JedisPool jedisPool = **new** JedisPool("192.168.25.128", 6379);  // 第二步：从JedisPool中获得Jedis对象。  Jedis jedis = jedisPool.getResource();  // 第三步：使用Jedis操作redis服务器。  jedis.set("jedis", "test");  String result = jedis.get("jedis");  System.***out***.println(result);  // 第四步：操作完毕后关闭jedis对象，连接池回收资源。  jedis.close();  // 第五步：关闭JedisPool对象。  jedisPool.close();  } |

##### 连接集群版

第一步：使用JedisCluster对象。需要一个Set<HostAndPort>参数。Redis节点的列表。

第二步：直接使用JedisCluster对象操作redis。在系统中单例存在。

第三步：打印结果

第四步：系统关闭前，关闭JedisCluster对象。

|  |
| --- |
| @Test  **public** **void** testJedisCluster() **throws** Exception {  // 第一步：使用JedisCluster对象。需要一个Set<HostAndPort>参数。Redis节点的列表。  Set<HostAndPort> nodes = **new** HashSet<>();  nodes.add(**new** HostAndPort("192.168.25.128", 7001));  nodes.add(**new** HostAndPort("192.168.25.128", 7002));  nodes.add(**new** HostAndPort("192.168.25.128", 7003));  nodes.add(**new** HostAndPort("192.168.25.128", 7004));  nodes.add(**new** HostAndPort("192.168.25.128", 7005));  nodes.add(**new** HostAndPort("192.168.25.128", 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。  jedisCluster.set("hello", "100");  String result = jedisCluster.get("hello");  // 第三步：打印结果  System.***out***.println(result);  // 第四步：系统关闭前，关闭JedisCluster对象。  jedisCluster.close();  } |

#### 向业务逻辑中添加缓存

##### 接口封装

常用的操作redis的方法提取出一个接口，分别对应单机版和集群版创建两个实现类。

###### 接口定义

|  |
| --- |
| **public** **interface** JedisClient {  String set(String key, String value);  String get(String key);  Boolean exists(String key);  Long expire(String key, **int** seconds);  Long ttl(String key);  Long incr(String key);  Long hset(String key, String field, String value);  String hget(String key, String field);  Long hdel(String key, String... field);  } |

###### 单机版实现类

|  |
| --- |
| **public** **class** JedisClientPool **implements** JedisClient {    @Autowired  **private** JedisPool jedisPool;  @Override  **public** String set(String key, String value) {  Jedis jedis = jedisPool.getResource();  String result = jedis.set(key, value);  jedis.close();  **return** result;  }  @Override  **public** String get(String key) {  Jedis jedis = jedisPool.getResource();  String result = jedis.get(key);  jedis.close();  **return** result;  }  @Override  **public** Boolean exists(String key) {  Jedis jedis = jedisPool.getResource();  Boolean result = jedis.exists(key);  jedis.close();  **return** result;  }  @Override  **public** Long expire(String key, **int** seconds) {  Jedis jedis = jedisPool.getResource();  Long result = jedis.expire(key, seconds);  jedis.close();  **return** result;  }  @Override  **public** Long ttl(String key) {  Jedis jedis = jedisPool.getResource();  Long result = jedis.ttl(key);  jedis.close();  **return** result;  }  @Override  **public** Long incr(String key) {  Jedis jedis = jedisPool.getResource();  Long result = jedis.incr(key);  jedis.close();  **return** result;  }  @Override  **public** Long hset(String key, String field, String value) {  Jedis jedis = jedisPool.getResource();  Long result = jedis.hset(key, field, value);  jedis.close();  **return** result;  }  @Override  **public** String hget(String key, String field) {  Jedis jedis = jedisPool.getResource();  String result = jedis.hget(key, field);  jedis.close();  **return** result;  }  @Override  **public** Long hdel(String key, String... field) {  Jedis jedis = jedisPool.getResource();  Long result = jedis.hdel(key, field);  jedis.close();  **return** result;  }  } |

配置：applicationContext-redis.xml

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:context=*"http://www.springframework.org/schema/context"* xmlns:p=*"http://www.springframework.org/schema/p"*  xmlns:aop=*"http://www.springframework.org/schema/aop"* xmlns:tx=*"http://www.springframework.org/schema/tx"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans4.2.xsd*  *http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context4.2.xsd*  *http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop4.2.xsd http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx4.2.xsd*  *http://www.springframework.org/schema/util http://www.springframework.org/schema/util/spring-util4.2.xsd"*>  <!-- 配置单机版的连接 -->  <bean id=*"jedisPool"* class=*"redis.clients.jedis.JedisPool"*>  <constructor-arg name=*"host"* value=*"192.168.25.128"*></constructor-arg>  <constructor-arg name=*"port"* value=*"6379"*></constructor-arg>  </bean>  <bean id=*"jedisClientPool"* class=*"cn.e3mall.jedis.JedisClientPool"*/>    </beans> |

###### 集群版实现类

|  |
| --- |
| **package** cn.e3mall.jedis;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** redis.clients.jedis.JedisCluster;  **public** **class** JedisClientCluster **implements** JedisClient {    @Autowired  **private** JedisCluster jedisCluster;  @Override  **public** String set(String key, String value) {  **return** jedisCluster.set(key, value);  }  @Override  **public** String get(String key) {  **return** jedisCluster.get(key);  }  @Override  **public** Boolean exists(String key) {  **return** jedisCluster.exists(key);  }  @Override  **public** Long expire(String key, **int** seconds) {  **return** jedisCluster.expire(key, seconds);  }  @Override  **public** Long ttl(String key) {  **return** jedisCluster.ttl(key);  }  @Override  **public** Long incr(String key) {  **return** jedisCluster.incr(key);  }  @Override  **public** Long hset(String key, String field, String value) {  **return** jedisCluster.hset(key, field, value);  }  @Override  **public** String hget(String key, String field) {  **return** jedisCluster.hget(key, field);  }  @Override  **public** Long hdel(String key, String... field) {  **return** jedisCluster.hdel(key, field);  }  } |

Spring的配置：

|  |
| --- |
| <!-- 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=*"192.168.25.128"*></constructor-arg>  <constructor-arg name=*"port"* value=*"7001"*></constructor-arg>  </bean>  <bean class=*"redis.clients.jedis.HostAndPort"*>  <constructor-arg name=*"host"* value=*"192.168.25.128"*></constructor-arg>  <constructor-arg name=*"port"* value=*"7002"*></constructor-arg>  </bean>  <bean class=*"redis.clients.jedis.HostAndPort"*>  <constructor-arg name=*"host"* value=*"192.168.25.128"*></constructor-arg>  <constructor-arg name=*"port"* value=*"7003"*></constructor-arg>  </bean>  <bean class=*"redis.clients.jedis.HostAndPort"*>  <constructor-arg name=*"host"* value=*"192.168.25.128"*></constructor-arg>  <constructor-arg name=*"port"* value=*"7004"*></constructor-arg>  </bean>  <bean class=*"redis.clients.jedis.HostAndPort"*>  <constructor-arg name=*"host"* value=*"192.168.25.128"*></constructor-arg>  <constructor-arg name=*"port"* value=*"7005"*></constructor-arg>  </bean>  <bean class=*"redis.clients.jedis.HostAndPort"*>  <constructor-arg name=*"host"* value=*"192.168.25.128"*></constructor-arg>  <constructor-arg name=*"port"* value=*"7006"*></constructor-arg>  </bean>  </set>  </constructor-arg>  </bean>  <bean id=*"jedisClientCluster"* class=*"cn.e3mall.jedis.JedisClientCluster"*/> |

注意：单机版和集群版不能共存，使用单机版时注释集群版的配置。使用集群版，把单机版注释。

##### 封装代码测试

|  |
| --- |
| @Test  **public** **void** testJedisClient() **throws** Exception {  //初始化Spring容器  ApplicationContext applicationContext = **new** ClassPathXmlApplicationContext("classpath:spring/applicationContext-redis.xml");  //从容器中获得JedisClient对象  JedisClient jedisClient = applicationContext.getBean(JedisClient.**class**);  jedisClient.set("first", "100");  String result = jedisClient.get("first");  System.***out***.println(result);    } |

### Redis的基本数据类型

#### 字符串类型(String)

|  |
| --- |
| redis 127.0.0.1:6379> SET mykey "redis"  OK  redis 127.0.0.1:6379> GET mykey  "redis" |

在上面的例子中，SET和GET是redis中的命令，而mykey是键的名称。

Redis字符串命令用于管理Redis中的字符串值。以下是使用Redis字符串命令的语法。

redis 127.0.0.1:6379> COMMAND KEY\_NAME

Shell

**示例**

redis 127.0.0.1:6379> SET mykey "redis"

OK

redis 127.0.0.1:6379> GET mykey

"redis"

Shell

在上面的例子中，SET和GET是redis中的命令，而mykey是键的名称。

Redis字符串命令

下表列出了一些用于在**Redis**中管理字符串的基本命令。

|  |  |  |
| --- | --- | --- |
| 编号 | 命令 | 描述说明 |
| 1 | [SET key value](http://www.yiibai.com/redis/strings_set.html) | 此命令设置指定键的值。 |
| 2 | [GET key](http://www.yiibai.com/redis/strings_get.html) | 获取指定键的值。 |
| 3 | [GETRANGE key start end](http://www.yiibai.com/redis/strings_getrange.html) | 获取存储在键上的字符串的子字符串。 |
| 4 | [GETSET key value](http://www.yiibai.com/redis/strings_getset.html) | 设置键的字符串值并返回其旧值。 |
| 5 | [GETBIT key offset](http://www.yiibai.com/redis/strings_getbit.html) | 返回在键处存储的字符串值中偏移处的位值。 |
| 6 | [MGET key1 [key2..]](http://www.yiibai.com/redis/strings_mget.html) | 获取所有给定键的值 |
| 7 | [SETBIT key offset value](http://www.yiibai.com/redis/strings_setbit.html) | 存储在键上的字符串值中设置或清除偏移处的位 |
| 8 | [SETEX key seconds value](http://www.yiibai.com/redis/strings_setex.html) | 使用键和到期时间来设置值 |
| 9 | [SETNX key value](http://www.yiibai.com/redis/strings_setnx.html) | 设置键的值，仅当键不存在时 |
| 10 | [SETRANGE key offset value](http://www.yiibai.com/redis/strings_setrange.html) | 在指定偏移处开始的键处覆盖字符串的一部分 |
| 11 | [STRLEN key](http://www.yiibai.com/redis/strings_strlen.html) | 获取存储在键中的值的长度 |
| 12 | [MSET key value [key value …]](http://www.yiibai.com/redis/strings_mset.html) | 为多个键分别设置它们的值 |
| 13 | [MSETNX key value [key value …]](http://www.yiibai.com/redis/strings_msetnx.html) | 为多个键分别设置它们的值，仅当键不存在时 |
| 14 | [PSETEX key milliseconds value](http://www.yiibai.com/redis/strings_psetex.html) | 设置键的值和到期时间(以毫秒为单位) |
| 15 | [INCR key](http://www.yiibai.com/redis/strings_incr.html) | 将键的整数值增加1 |
| 16 | [INCRBY key increment](http://www.yiibai.com/redis/strings_incrby.html) | 将键的整数值按给定的数值增加 |
| 17 | [INCRBYFLOAT key increment](http://www.yiibai.com/redis/strings_incrbyfloat.html) | 将键的浮点值按给定的数值增加 |
| 18 | [DECR key](http://www.yiibai.com/redis/strings_decr.html) | 将键的整数值减1 |
| 19 | [DECRBY key decrement](http://www.yiibai.com/redis/strings_decrby.html) | 按给定数值减少键的整数值 |
| 20 | [APPEND key value](http://www.yiibai.com/redis/strings_append.html) | 将指定值附加到键 |

#### 列表类型(List)

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)。

redis 127.0.0.1:6379> LPUSH runoobkey redis

(integer) 1

redis 127.0.0.1:6379> LPUSH runoobkey mongodb

(integer) 2

redis 127.0.0.1:6379> LPUSH runoobkey mysql

(integer) 3

redis 127.0.0.1:6379> LRANGE runoobkey 0 10

1) "mysql"

2) "mongodb"

3) "redis"

Redis 列表命令

下表列出了列表相关的基本命令：

|  |  |
| --- | --- |
| 序号 | 命令及描述 |
| 1 | [BLPOP key1 [key2 ] timeout](http://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。 |
| 2 | [BRPOP key1 [key2 ] timeout](http://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。 |
| 3 | [BRPOPLPUSH source destination timeout](http://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。 |
| 4 | [LINDEX key index](http://www.runoob.com/redis/lists-lindex.html)  通过索引获取列表中的元素 |
| 5 | [LINSERT key BEFORE|AFTER pivot value](http://www.runoob.com/redis/lists-linsert.html)  在列表的元素前或者后插入元素 |
| 6 | [LLEN key](http://www.runoob.com/redis/lists-llen.html)  获取列表长度 |
| 7 | [LPOP key](http://www.runoob.com/redis/lists-lpop.html)  移出并获取列表的第一个元素 |
| 8 | [LPUSH key value1 [value2]](http://www.runoob.com/redis/lists-lpush.html)  将一个或多个值插入到列表头部 |
| 9 | [LPUSHX key value](http://www.runoob.com/redis/lists-lpushx.html)  将一个值插入到已存在的列表头部 |
| 10 | [LRANGE key start stop](http://www.runoob.com/redis/lists-lrange.html)  获取列表指定范围内的元素 |
| 11 | [LREM key count value](http://www.runoob.com/redis/lists-lrem.html)  移除列表元素 |
| 12 | [LSET key index value](http://www.runoob.com/redis/lists-lset.html)  通过索引设置列表元素的值 |
| 13 | [LTRIM key start stop](http://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。 |
| 14 | [RPOP key](http://www.runoob.com/redis/lists-rpop.html)  移除并获取列表最后一个元素 |
| 15 | [RPOPLPUSH source destination](http://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 |
| 16 | [RPUSH key value1 [value2]](http://www.runoob.com/redis/lists-rpush.html)  在列表中添加一个或多个值 |
| 17 | [RPUSHX key value](http://www.runoob.com/redis/lists-rpushx.html)  为已存在的列表添加值 |

#### Redis 集合(Set)

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)。

**实例**

redis 127.0.0.1:6379> SADD runoobkey redis

(integer) 1

redis 127.0.0.1:6379> SADD runoobkey mongodb

(integer) 1

redis 127.0.0.1:6379> SADD runoobkey mysql

(integer) 1

redis 127.0.0.1:6379> SADD runoobkey mysql

(integer) 0

redis 127.0.0.1:6379> SMEMBERS runoobkey

1) "mysql"

2) "mongodb"

3) "redis"

在以上实例中我们通过 **SADD** 命令向名为 **runoobkey** 的集合插入的三个元素。

**Redis 集合命令**

下表列出了 Redis 集合基本命令：

|  |  |
| --- | --- |
| **序号** | **命令及描述** |
| 1 | [SADD key member1 [member2]](http://www.runoob.com/redis/sets-sadd.html)  向集合添加一个或多个成员 |
| 2 | [SCARD key](http://www.runoob.com/redis/sets-scard.html)  获取集合的成员数 |
| 3 | [SDIFF key1 [key2]](http://www.runoob.com/redis/sets-sdiff.html)  返回给定所有集合的差集 |
| 4 | [SDIFFSTORE destination key1 [key2]](http://www.runoob.com/redis/sets-sdiffstore.html)  返回给定所有集合的差集并存储在 destination 中 |
| 5 | [SINTER key1 [key2]](http://www.runoob.com/redis/sets-sinter.html)  返回给定所有集合的交集 |
| 6 | [SINTERSTORE destination key1 [key2]](http://www.runoob.com/redis/sets-sinterstore.html)  返回给定所有集合的交集并存储在 destination 中 |
| 7 | [SISMEMBER key member](http://www.runoob.com/redis/sets-sismember.html)  判断 member 元素是否是集合 key 的成员 |
| 8 | [SMEMBERS key](http://www.runoob.com/redis/sets-smembers.html)  返回集合中的所有成员 |
| 9 | [SMOVE source destination member](http://www.runoob.com/redis/sets-smove.html)  将 member 元素从 source 集合移动到 destination 集合 |
| 10 | [SPOP key](http://www.runoob.com/redis/sets-spop.html)  移除并返回集合中的一个随机元素 |
| 11 | [SRANDMEMBER key [count]](http://www.runoob.com/redis/sets-srandmember.html)  返回集合中一个或多个随机数 |
| 12 | [SREM key member1 [member2]](http://www.runoob.com/redis/sets-srem.html)  移除集合中一个或多个成员 |
| 13 | [SUNION key1 [key2]](http://www.runoob.com/redis/sets-sunion.html)  返回所有给定集合的并集 |
| 14 | [SUNIONSTORE destination key1 [key2]](http://www.runoob.com/redis/sets-sunionstore.html)  所有给定集合的并集存储在 destination 集合中 |
| 15 | [SSCAN key cursor [MATCH pattern] [COUNT count]](http://www.runoob.com/redis/sets-sscan.html)  迭代集合中的元素 |

#### Redis 有序集合(sorted set)

Redis 有序集合和集合一样也是string类型元素的集合,且不允许重复的成员。

不同的是每个元素都会关联一个double类型的分数。redis正是通过分数来为集合中的成员进行从小到大的排序。

有序集合的成员是唯一的,但分数(score)却可以重复。

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)。

**实例**

redis 127.0.0.1:6379> ZADD runoobkey 1 redis

(integer) 1

redis 127.0.0.1:6379> ZADD runoobkey 2 mongodb

(integer) 1

redis 127.0.0.1:6379> ZADD runoobkey 3 mysql

(integer) 1

redis 127.0.0.1:6379> ZADD runoobkey 3 mysql

(integer) 0

redis 127.0.0.1:6379> ZADD runoobkey 4 mysql

(integer) 0

redis 127.0.0.1:6379> ZRANGE runoobkey 0 10 WITHSCORES

1) "redis"

2) "1"

3) "mongodb"

4) "2"

5) "mysql"

6) "4"

在以上实例中我们通过命令 **ZADD** 向 redis 的有序集合中添加了三个值并关联上分数。

**Redis 有序集合命令**

下表列出了 redis 有序集合的基本命令:

|  |  |
| --- | --- |
| **序号** | **命令及描述** |
| 1 | [ZADD key score1 member1 [score2 member2]](http://www.runoob.com/redis/sorted-sets-zadd.html)  向有序集合添加一个或多个成员，或者更新已存在成员的分数 |
| 2 | [ZCARD key](http://www.runoob.com/redis/sorted-sets-zcard.html)  获取有序集合的成员数 |
| 3 | [ZCOUNT key min max](http://www.runoob.com/redis/sorted-sets-zcount.html)  计算在有序集合中指定区间分数的成员数 |
| 4 | [ZINCRBY key increment member](http://www.runoob.com/redis/sorted-sets-zincrby.html)  有序集合中对指定成员的分数加上增量 increment |
| 5 | [ZINTERSTORE destination numkeys key [key ...]](http://www.runoob.com/redis/sorted-sets-zinterstore.html)  计算给定的一个或多个有序集的交集并将结果集存储在新的有序集合 key 中 |
| 6 | [ZLEXCOUNT key min max](http://www.runoob.com/redis/sorted-sets-zlexcount.html)  在有序集合中计算指定字典区间内成员数量 |
| 7 | [ZRANGE key start stop [WITHSCORES]](http://www.runoob.com/redis/sorted-sets-zrange.html)  通过索引区间返回有序集合成指定区间内的成员 |
| 8 | [ZRANGEBYLEX key min max [LIMIT offset count]](http://www.runoob.com/redis/sorted-sets-zrangebylex.html)  通过字典区间返回有序集合的成员 |
| 9 | [ZRANGEBYSCORE key min max [WITHSCORES] [LIMIT]](http://www.runoob.com/redis/sorted-sets-zrangebyscore.html)  通过分数返回有序集合指定区间内的成员 |
| 10 | [ZRANK key member](http://www.runoob.com/redis/sorted-sets-zrank.html)  返回有序集合中指定成员的索引 |
| 11 | [ZREM key member [member ...]](http://www.runoob.com/redis/sorted-sets-zrem.html)  移除有序集合中的一个或多个成员 |
| 12 | [ZREMRANGEBYLEX key min max](http://www.runoob.com/redis/sorted-sets-zremrangebylex.html)  移除有序集合中给定的字典区间的所有成员 |
| 13 | [ZREMRANGEBYRANK key start stop](http://www.runoob.com/redis/sorted-sets-zremrangebyrank.html)  移除有序集合中给定的排名区间的所有成员 |
| 14 | [ZREMRANGEBYSCORE key min max](http://www.runoob.com/redis/sorted-sets-zremrangebyscore.html)  移除有序集合中给定的分数区间的所有成员 |
| 15 | [ZREVRANGE key start stop [WITHSCORES]](http://www.runoob.com/redis/sorted-sets-zrevrange.html)  返回有序集中指定区间内的成员，通过索引，分数从高到底 |
| 16 | [ZREVRANGEBYSCORE key max min [WITHSCORES]](http://www.runoob.com/redis/sorted-sets-zrevrangebyscore.html)  返回有序集中指定分数区间内的成员，分数从高到低排序 |
| 17 | [ZREVRANK key member](http://www.runoob.com/redis/sorted-sets-zrevrank.html)  返回有序集合中指定成员的排名，有序集成员按分数值递减(从大到小)排序 |
| 18 | [ZSCORE key member](http://www.runoob.com/redis/sorted-sets-zscore.html)  返回有序集中，成员的分数值 |
| 19 | [ZUNIONSTORE destination numkeys key [key ...]](http://www.runoob.com/redis/sorted-sets-zunionstore.html)  计算给定的一个或多个有序集的并集，并存储在新的 key 中 |
| 20 | [ZSCAN key cursor [MATCH pattern] [COUNT count]](http://www.runoob.com/redis/sorted-sets-zscan.html)  迭代有序集合中的元素（包括元素成员和元素分值） |

#### Redis 哈希(Hash)

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）。

**实例**

127.0.0.1:6379> HMSET runoobkey name "redis tutorial" description "redis basic commands for caching" likes 20 visitors 23000

OK

127.0.0.1:6379> HGETALL runoobkey

1) "name"

2) "redis tutorial"

3) "description"

4) "redis basic commands for caching"

5) "likes"

6) "20"

7) "visitors"

8) "23000"

**hset key mapHey MapValue**

在以上实例中，我们设置了 redis 的一些描述信息(name, description, likes, visitors) 到哈希表的 **runoobkey** 中。

**Redis hash 命令**

下表列出了 redis hash 基本的相关命令：

|  |  |
| --- | --- |
| **序号** | **命令及描述** |
| 1 | [HDEL key field2 [field2]](http://www.runoob.com/redis/hashes-hdel.html)  删除一个或多个哈希表字段 |
| 2 | [HEXISTS key field](http://www.runoob.com/redis/hashes-hexists.html)  查看哈希表 key 中，指定的字段是否存在。 |
| 3 | [HGET key field](http://www.runoob.com/redis/hashes-hget.html)  获取存储在哈希表中指定字段的值。 |
| 4 | [HGETALL key](http://www.runoob.com/redis/hashes-hgetall.html)  获取在哈希表中指定 key 的所有字段和值 |
| 5 | [HINCRBY key field increment](http://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 。 |
| 6 | [HINCRBYFLOAT key field increment](http://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 。 |
| 7 | [HKEYS key](http://www.runoob.com/redis/hashes-hkeys.html)  获取所有哈希表中的字段 |
| 8 | [HLEN key](http://www.runoob.com/redis/hashes-hlen.html)  获取哈希表中字段的数量 |
| 9 | [HMGET key field1 [field2]](http://www.runoob.com/redis/hashes-hmget.html)  获取所有给定字段的值 |
| 10 | [HMSET key field1 value1 [field2 value2 ]](http://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。 |
| 11 | [HSET key field value](http://www.runoob.com/redis/hashes-hset.html)  将哈希表 key 中的字段 field 的值设为 value 。 |
| 12 | [HSETNX key field value](http://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。 |
| 13 | [HVALS key](http://www.runoob.com/redis/hashes-hvals.html)  获取哈希表中所有值 |
| 14 | HSCAN key cursor [MATCH pattern] [COUNT count]  迭代哈希表中的键值对。 |

## 实战

### redis配置

设置redis配置

* protected-mode no
* bind 127.0.0.1

### Redis jedis

#### 引入Maven依赖

|  |
| --- |
| <!-- https://mvnrepository.com/artifact/redis.clients/jedis -->  <dependency>  <groupId>redis.clients</groupId>  <artifactId>jedis</artifactId>  <version>2.9.0</version>  </dependency> |

#### Java代码

##### JedisUtils

|  |
| --- |
| package com.yaosiyuan.redis;  import redis.clients.jedis.Jedis;  import redis.clients.jedis.JedisPool;  import redis.clients.jedis.JedisPoolConfig;  public class JedisUtils {  // 创建连接池  private static JedisPoolConfig config;  private static JedisPool pool;  static {  config = new JedisPoolConfig();  config.setMaxTotal(30);  config.setMaxIdle(2);  pool = new JedisPool(config, "47.94.238.23", 6379);  }  // 获取连接的方法  public static Jedis getJedis() {  return pool.getResource();  }  // 释放连接  public static void closeJedis(Jedis j) {  j.close();  }  } |

##### TestRedis01

|  |
| --- |
| package com.yaosiyuan.redis;  import java.util.HashMap;  import java.util.Iterator;  import java.util.List;  import java.util.Map;  import org.junit.Before;  import org.junit.Test;  import redis.clients.jedis.Jedis;  public class TestRedis01 {  private Jedis jedis;  public static void main(String[] args) {  // 连接本地的 Redis 服务  Jedis jedis = new Jedis("47.94.238.23");  System.out.println("连接成功");  // 设置 redis 字符串数据  jedis.set("key1", "value1");  // 获取存储的数据并输出  System.out.println("redis 存储的字符串为: " + jedis.get("key1"));  }  @Before  public void setup() {  // 连接redis服务器，192.168.0.100:6379  jedis = new Jedis("47.94.238.23", 6379);  // 权限认证  // jedis.auth("admin");  }  /\*\*  \* redis存储字符串  \*/  @Test  public void testString() {  // -----添加数据----------  jedis.set("name", "xinxin");// 向key-->name中放入了value-->xinxin  System.out.println(jedis.get("name"));// 执行结果：xinxin  jedis.append("name", " is my lover"); // 拼接  System.out.println(jedis.get("name"));  jedis.del("name"); // 删除某个键  System.out.println(jedis.get("name"));  // 设置多个键值对  jedis.mset("name", "liuling", "age", "23", "qq", "476777XXX");  jedis.incr("age"); // 进行加1操作  System.out.println(jedis.get("name") + "-" + jedis.get("age") + "-" + jedis.get("qq"));  }  /\*\*  \* redis操作Map  \*/  @Test  public void testMap() {  // -----添加数据----------  Map<String, String> map = new HashMap<String, String>();  map.put("name", "xinxin");  map.put("age", "22");  map.put("qq", "123456");  jedis.hmset("user", map);  // 取出user中的name，执行结果:[minxr]-->注意结果是一个泛型的List  // 第一个参数是存入redis中map对象的key，后面跟的是放入map中的对象的key，后面的key可以跟多个，是可变参数  List<String> rsmap = jedis.hmget("user", "name", "age", "qq");  System.out.println(rsmap);  // 删除map中的某个键值  jedis.hdel("user", "age");  System.out.println(jedis.hmget("user", "age")); // 因为删除了，所以返回的是null  System.out.println(jedis.hlen("user")); // 返回key为user的键中存放的值的个数2  System.out.println(jedis.exists("user"));// 是否存在key为user的记录 返回true  System.out.println(jedis.hkeys("user"));// 返回map对象中的所有key  System.out.println(jedis.hvals("user"));// 返回map对象中的所有value  Iterator<String> iter = jedis.hkeys("user").iterator();  while (iter.hasNext()) {  String key = iter.next();  System.out.println(key + ":" + jedis.hmget("user", key));  }  }  /\*\*  \* jedis操作List  \*/  @Test  public void testList() {  // 开始前，先移除所有的内容  jedis.del("java framework");  System.out.println(jedis.lrange("java framework", 0, -1));  // 先向key java framework中存放三条数据  jedis.lpush("java framework", "spring");  jedis.lpush("java framework", "struts");  jedis.lpush("java framework", "hibernate");  // 再取出所有数据jedis.lrange是按范围取出，  // 第一个是key，第二个是起始位置，第三个是结束位置，jedis.llen获取长度 -1表示取得所有  System.out.println(jedis.lrange("java framework", 0, -1));  jedis.del("java framework");  jedis.rpush("java framework", "spring");  jedis.rpush("java framework", "struts");  jedis.rpush("java framework", "hibernate");  System.out.println(jedis.lrange("java framework", 0, -1));  }  /\*\*  \* jedis操作Set  \*/  @Test  public void testSet() {  // 添加  jedis.sadd("user", "liuling");  jedis.sadd("user", "xinxin");  jedis.sadd("user", "ling");  jedis.sadd("user", "zhangxinxin");  jedis.sadd("user", "who");  // 移除noname  jedis.srem("user", "who");  System.out.println(jedis.smembers("user"));// 获取所有加入的value  System.out.println(jedis.sismember("user", "who"));// 判断 who  // 是否是user集合的元素  System.out.println(jedis.srandmember("user"));  System.out.println(jedis.scard("user"));// 返回集合的元素个数  }  @Test  public void test() throws InterruptedException {  // jedis 排序  // 注意，此处的rpush和lpush是List的操作。是一个双向链表（但从表现来看的）  jedis.del("a");// 先清除数据，再加入数据进行测试  jedis.rpush("a", "1");  jedis.lpush("a", "6");  jedis.lpush("a", "3");  jedis.lpush("a", "9");  System.out.println(jedis.lrange("a", 0, -1));// [9, 3, 6, 1]  System.out.println(jedis.sort("a")); // [1, 3, 6, 9] //输入排序后结果  System.out.println(jedis.lrange("a", 0, -1));  }  @Test  public void testRedisPool() {  JedisUtils.getJedis().set("newname", "中文测试");  System.out.println(JedisUtils.getJedis().get("newname"));  }  } |

#### SpringBoot集成Redis

#### 引入maven依赖

|  |
| --- |
| <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.5.3.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-data-redis</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  </dependencies> |

#### 新增配置文件信息

springboot 1.5.x 版本

|  |
| --- |
| ########################################################  ###Redis (RedisConfiguration)  ########################################################  spring.redis.database=0  spring.redis.host=127.0.0.1  spring.redis.port=6379  spring.redis.password=123456  spring.redis.pool.max-idle=8  spring.redis.pool.min-idle=0  spring.redis.pool.max-active=8  spring.redis.pool.max-wait=-1  spring.redis.timeout=5000 |

|  |
| --- |
| spring:  cache:  type: redis  redis:  host: 127.0.0.1  port: 6379  timeout: 0  database: 0  pool:  max-active: 8  max-wait: -1  max-idle: 8  min-idle: 0 |

springboot 2.x 版本

|  |
| --- |
| #2.x版本中由于引入了不同客户端，需要指定配置哪种连接池  #jedis客户端  spring:  cache:  type: redis  redis:  host: 127.0.0.1  port: 6379  timeout: 0  database: 0  jedis:  pool:  max-active: 8  max-wait: -1ms  max-idle: 8  min-idle: 0 |

#### Java代码

|  |
| --- |
| @Service  public class RedisService {  @Autowired  private StringRedisTemplate stringRedisTemplate;  public void setStr(String key, String value) {  setStr(key, value, null);  }  public void setStr(String key, String value, Long time) {  stringRedisTemplate.opsForValue().set(key, value);  if (time != null)  stringRedisTemplate.expire(key, time, TimeUnit.*SECONDS*);  }  public Object getKey(String key) {  return stringRedisTemplate.opsForValue().get(key);  }  public void delKey(String key) {  stringRedisTemplate.delete(key);  }  } |

### spring集成redis

#### 依赖

|  |
| --- |
| <dependency>  <groupId>redis.clients</groupId>  <artifactId>jedis</artifactId>  <version>2.9.0</version>  </dependency>  <!-- redis Spring 基于注解配置 -->  <dependency>  <groupId>org.springframework.data</groupId>  <artifactId>spring-data-redis</artifactId>  <version>1.7.2.RELEASE</version>  </dependency>  <dependency>  <groupId>com.fasterxml.jackson.core</groupId>  <artifactId>jackson-core</artifactId>  <version>2.7.5</version>  </dependency>  <dependency>  <groupId>com.fasterxml.jackson.core</groupId>  <artifactId>jackson-annotations</artifactId>  <version>2.7.5</version>  </dependency>  <dependency>  <groupId>com.fasterxml.jackson.core</groupId>  <artifactId>jackson-databind</artifactId>  <version>2.7.5</version>  </dependency>  <dependency>  <groupId>org.codehaus.jackson</groupId>  <artifactId>jackson-mapper-asl</artifactId>  <version>1.9.4</version>  </dependency>    <dependency>  <groupId>org.codehaus.jackson</groupId>  <artifactId>jackson-core-asl</artifactId>  <version>1.9.4</version>  </dependency> |

#### redis.properties

|  |
| --- |
| #ip地址  redis.hostName=192.168.253.142  #端口号  redis.port=6379  #如果有密码  redis.password=  #客户端超时时间单位是毫秒 默认是2000  redis.timeout=10000      #最大空闲数  redis.maxIdle=300  #连接池的最大数据库连接数。设为0表示无限制,如果是jedis 2.4以后用redis.maxTotal  #redis.maxActive=600  #控制一个pool可分配多少个jedis实例,用来替换上面的redis.maxActive,如果是jedis 2.4以后用该属性  redis.maxTotal=1000  #最大建立连接等待时间。如果超过此时间将接到异常。设为-1表示无限制。  redis.maxWaitMillis=1000  #连接的最小空闲时间 默认1800000毫秒(30分钟)  redis.minEvictableIdleTimeMillis=300000  #每次释放连接的最大数目,默认3  redis.numTestsPerEvictionRun=1024  #逐出扫描的时间间隔(毫秒) 如果为负数,则不运行逐出线程, 默认-1  redis.timeBetweenEvictionRunsMillis=30000  #是否在从池中取出连接前进行检验,如果检验失败,则从池中去除连接并尝试取出另一个  redis.testOnBorrow=true  #在空闲时检查有效性, 默认false  redis.testWhileIdle=true |

#### applicationContext-redis.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:p="http://www.springframework.org/schema/p"  xmlns:context="http://www.springframework.org/schema/context"  xmlns:mvc="http://www.springframework.org/schema/mvc"  xmlns:cache="http://www.springframework.org/schema/cache"  xsi:schemaLocation="http://www.springframework.org/schema/beans  http://www.springframework.org/schema/beans/spring-beans-4.2.xsd  http://www.springframework.org/schema/context  http://www.springframework.org/schema/context/spring-context-4.2.xsd  http://www.springframework.org/schema/mvc  http://www.springframework.org/schema/mvc/spring-mvc-4.2.xsd  http://www.springframework.org/schema/cache  http://www.springframework.org/schema/cache/spring-cache-4.2.xsd">    <!-- 加载配置文件 -->  <context:property-placeholder location="classpath:\*.properties" />  <!-- redis连接池配置-->  <bean id="jedisPoolConfig" class="redis.clients.jedis.JedisPoolConfig" >  <!--最大空闲数-->  <property name="maxIdle" value="${redis.maxIdle}" />  <!--连接池的最大数据库连接数 -->  <property name="maxTotal" value="${redis.maxTotal}" />  <!--最大建立连接等待时间-->  <property name="maxWaitMillis" value="${redis.maxWaitMillis}" />  <!--逐出连接的最小空闲时间 默认1800000毫秒(30分钟)-->  <property name="minEvictableIdleTimeMillis" value="${redis.minEvictableIdleTimeMillis}" />  <!--每次逐出检查时 逐出的最大数目 如果为负数就是 : 1/abs(n), 默认3-->  <property name="numTestsPerEvictionRun" value="${redis.numTestsPerEvictionRun}" />  <!--逐出扫描的时间间隔(毫秒) 如果为负数,则不运行逐出线程, 默认-1-->  <property name="timeBetweenEvictionRunsMillis" value="${redis.timeBetweenEvictionRunsMillis}" />  <!--是否在从池中取出连接前进行检验,如果检验失败,则从池中去除连接并尝试取出另一个-->  <property name="testOnBorrow" value="${redis.testOnBorrow}" />  <!--在空闲时检查有效性, 默认false -->  <property name="testWhileIdle" value="${redis.testWhileIdle}" />  </bean >        <!--redis连接工厂 -->  <bean id="jedisConnectionFactory" class="org.springframework.data.redis.connection.jedis.JedisConnectionFactory" destroy-method="destroy">  <property name="poolConfig" ref="jedisPoolConfig"></property>  <!--IP地址 -->  <property name="hostName" value="${redis.hostName}"></property>  <!--端口号 -->  <property name="port" value="${redis.port}"></property>  <!--如果Redis设置有密码 -->  <property name="password" value="${redis.password}" />  <!--客户端超时时间单位是毫秒 -->  <property name="timeout" value="${redis.timeout}"></property>  </bean>  <!--redis操作模版,使用该对象可以操作redis -->  <bean id="redisTemplate" class="org.springframework.data.redis.core.RedisTemplate" >  <property name="connectionFactory" ref="jedisConnectionFactory" />  <!--如果不配置Serializer，那么存储的时候缺省使用String，如果用User类型存储，那么会提示错误User can't cast to String！！ -->  <property name="keySerializer" >  <bean class="org.springframework.data.redis.serializer.StringRedisSerializer" />  </property>  <property name="valueSerializer" >  <bean class="org.springframework.data.redis.serializer.GenericJackson2JsonRedisSerializer" />  </property>  <property name="hashKeySerializer">  <bean class="org.springframework.data.redis.serializer.StringRedisSerializer"/>  </property>  <property name="hashValueSerializer">  <bean class="org.springframework.data.redis.serializer.GenericJackson2JsonRedisSerializer"/>  </property>  <!--开启事务 -->  <property name="enableTransactionSupport" value="true"></property>  </bean >    <!--自定义redis工具类,在需要缓存的地方注入此类 -->  <bean id="redisUtil" class="com.yaosiyuan.redis.RedisUtil">  <property name="redisTemplate" ref="redisTemplate" />  </bean>  </beans> |

#### RedisUtil

|  |
| --- |
| package com.yaosiyuan.redis;  import java.util.List;  import java.util.Map;  import java.util.Set;  import java.util.concurrent.TimeUnit;  import org.springframework.data.redis.core.RedisTemplate;  import org.springframework.util.CollectionUtils;  /\*\*  \* 基于spring和redis的redisTemplate工具类 针对所有的hash 都是以h开头的方法 针对所有的Set 都是以s开头的方法 不含通用方法  \* 针对所有的List 都是以l开头的方法  \*/  public class RedisUtil {  private RedisTemplate<String, Object> redisTemplate;  public void setRedisTemplate(RedisTemplate<String, Object> redisTemplate) {  this.redisTemplate = redisTemplate;  }  // =============================common============================  /\*\*  \* 指定缓存失效时间  \*  \* @param key  \* 键  \* @param time  \* 时间(秒)  \* @return  \*/  public boolean expire(String key, long time) {  try {  if (time > 0) {  redisTemplate.expire(key, time, TimeUnit.SECONDS);  }  return true;  } catch (Exception e) {  e.printStackTrace();  return false;  }  }  /\*\*  \* 根据key 获取过期时间  \*  \* @param key  \* 键 不能为null  \* @return 时间(秒) 返回0代表为永久有效  \*/  public long getExpire(String key) {  return redisTemplate.getExpire(key, TimeUnit.SECONDS);  }  /\*\*  \* 判断key是否存在  \*  \* @param key  \* 键  \* @return true 存在 false不存在  \*/  public boolean hasKey(String key) {  try {  return redisTemplate.hasKey(key);  } catch (Exception e) {  e.printStackTrace();  return false;  }  }  /\*\*  \* 删除缓存  \*  \* @param key  \* 可以传一个值 或多个  \*/  @SuppressWarnings("unchecked")  public void del(String... key) {  if (key != null && key.length > 0) {  if (key.length == 1) {  redisTemplate.delete(key[0]);  } else {  redisTemplate.delete(CollectionUtils.arrayToList(key));  }  }  }  // ============================String=============================  /\*\*  \* 普通缓存获取  \*  \* @param key  \* 键  \* @return 值  \*/  public Object get(String key) {  return key == null ? null : redisTemplate.opsForValue().get(key);  }  /\*\*  \* 普通缓存放入  \*  \* @param key  \* 键  \* @param value  \* 值  \* @return true成功 false失败  \*/  public boolean set(String key, Object value) {  try {  redisTemplate.opsForValue().set(key, value);  return true;  } catch (Exception e) {  e.printStackTrace();  return false;  }  }  /\*\*  \* 普通缓存放入并设置时间  \*  \* @param key  \* 键  \* @param value  \* 值  \* @param time  \* 时间(秒) time要大于0 如果time小于等于0 将设置无限期  \* @return true成功 false 失败  \*/  public boolean set(String key, Object value, long time) {  try {  if (time > 0) {  redisTemplate.opsForValue().set(key, value, time, TimeUnit.SECONDS);  } else {  set(key, value);  }  return true;  } catch (Exception e) {  e.printStackTrace();  return false;  }  }  /\*\*  \* 递增  \*  \* @param key  \* 键  \* @param by  \* 要增加几(大于0)  \* @return  \*/  public long incr(String key, long delta) {  if (delta < 0) {  throw new RuntimeException("递增因子必须大于0");  }  return redisTemplate.opsForValue().increment(key, delta);  }  /\*\*  \* 递减  \*  \* @param key  \* 键  \* @param by  \* 要减少几(小于0)  \* @return  \*/  public long decr(String key, long delta) {  if (delta < 0) {  throw new RuntimeException("递减因子必须大于0");  }  return redisTemplate.opsForValue().increment(key, -delta);  }  // ================================Map=================================  /\*\*  \* HashGet  \*  \* @param key  \* 键 不能为null  \* @param item  \* 项 不能为null  \* @return 值  \*/  public Object hget(String key, String item) {  return redisTemplate.opsForHash().get(key, item);  }  /\*\*  \* 获取hashKey对应的所有键值  \*  \* @param key  \* 键  \* @return 对应的多个键值  \*/  public Map<Object, Object> hmget(String key) {  return redisTemplate.opsForHash().entries(key);  }  /\*\*  \* HashSet  \*  \* @param key  \* 键  \* @param map  \* 对应多个键值  \* @return true 成功 false 失败  \*/  public boolean hmset(String key, Map<String, Object> map) {  try {  redisTemplate.opsForHash().putAll(key, map);  return true;  } catch (Exception e) {  e.printStackTrace();  return false;  }  }  /\*\*  \* HashSet 并设置时间  \*  \* @param key  \* 键  \* @param map  \* 对应多个键值  \* @param time  \* 时间(秒)  \* @return true成功 false失败  \*/  public boolean hmset(String key, Map<String, Object> map, long time) {  try {  redisTemplate.opsForHash().putAll(key, map);  if (time > 0) {  expire(key, time);  }  return true;  } catch (Exception e) {  e.printStackTrace();  return false;  }  }  /\*\*  \* 向一张hash表中放入数据,如果不存在将创建  \*  \* @param key  \* 键  \* @param item  \* 项  \* @param value  \* 值  \* @return true 成功 false失败  \*/  public boolean hset(String key, String item, Object value) {  try {  redisTemplate.opsForHash().put(key, item, value);  return true;  } catch (Exception e) {  e.printStackTrace();  return false;  }  }  /\*\*  \* 向一张hash表中放入数据,如果不存在将创建  \*  \* @param key  \* 键  \* @param item  \* 项  \* @param value  \* 值  \* @param time  \* 时间(秒) 注意:如果已存在的hash表有时间,这里将会替换原有的时间  \* @return true 成功 false失败  \*/  public boolean hset(String key, String item, Object value, long time) {  try {  redisTemplate.opsForHash().put(key, item, value);  if (time > 0) {  expire(key, time);  }  return true;  } catch (Exception e) {  e.printStackTrace();  return false;  }  }  /\*\*  \* 删除hash表中的值  \*  \* @param key  \* 键 不能为null  \* @param item  \* 项 可以使多个 不能为null  \*/  public void hdel(String key, Object... item) {  redisTemplate.opsForHash().delete(key, item);  }  /\*\*  \* 判断hash表中是否有该项的值  \*  \* @param key  \* 键 不能为null  \* @param item  \* 项 不能为null  \* @return true 存在 false不存在  \*/  public boolean hHasKey(String key, String item) {  return redisTemplate.opsForHash().hasKey(key, item);  }  /\*\*  \* hash递增 如果不存在,就会创建一个 并把新增后的值返回  \*  \* @param key  \* 键  \* @param item  \* 项  \* @param by  \* 要增加几(大于0)  \* @return  \*/  public double hincr(String key, String item, double by) {  return redisTemplate.opsForHash().increment(key, item, by);  }  /\*\*  \* hash递减  \*  \* @param key  \* 键  \* @param item  \* 项  \* @param by  \* 要减少记(小于0)  \* @return  \*/  public double hdecr(String key, String item, double by) {  return redisTemplate.opsForHash().increment(key, item, -by);  }  // ============================set=============================  /\*\*  \* 根据key获取Set中的所有值  \*  \* @param key  \* 键  \* @return  \*/  public Set<Object> sGet(String key) {  try {  return redisTemplate.opsForSet().members(key);  } catch (Exception e) {  e.printStackTrace();  return null;  }  }  /\*\*  \* 根据value从一个set中查询,是否存在  \*  \* @param key  \* 键  \* @param value  \* 值  \* @return true 存在 false不存在  \*/  public boolean sHasKey(String key, Object value) {  try {  return redisTemplate.opsForSet().isMember(key, value);  } catch (Exception e) {  e.printStackTrace();  return false;  }  }  /\*\*  \* 将数据放入set缓存  \*  \* @param key  \* 键  \* @param values  \* 值 可以是多个  \* @return 成功个数  \*/  public long sSet(String key, Object... values) {  try {  return redisTemplate.opsForSet().add(key, values);  } catch (Exception e) {  e.printStackTrace();  return 0;  }  }  /\*\*  \* 将set数据放入缓存  \*  \* @param key  \* 键  \* @param time  \* 时间(秒)  \* @param values  \* 值 可以是多个  \* @return 成功个数  \*/  public long sSetAndTime(String key, long time, Object... values) {  try {  Long count = redisTemplate.opsForSet().add(key, values);  if (time > 0)  expire(key, time);  return count;  } catch (Exception e) {  e.printStackTrace();  return 0;  }  }  /\*\*  \* 获取set缓存的长度  \*  \* @param key  \* 键  \* @return  \*/  public long sGetSetSize(String key) {  try {  return redisTemplate.opsForSet().size(key);  } catch (Exception e) {  e.printStackTrace();  return 0;  }  }  /\*\*  \* 移除值为value的  \*  \* @param key  \* 键  \* @param values  \* 值 可以是多个  \* @return 移除的个数  \*/  public long setRemove(String key, Object... values) {  try {  Long count = redisTemplate.opsForSet().remove(key, values);  return count;  } catch (Exception e) {  e.printStackTrace();  return 0;  }  }  // ===============================list=================================  /\*\*  \* 获取list缓存的内容  \*  \* @param key  \* 键  \* @param start  \* 开始  \* @param end  \* 结束 0 到 -1代表所有值  \* @return  \*/  public List<Object> lGet(String key, long start, long end) {  try {  return redisTemplate.opsForList().range(key, start, end);  } catch (Exception e) {  e.printStackTrace();  return null;  }  }  /\*\*  \* 获取list缓存的长度  \*  \* @param key  \* 键  \* @return  \*/  public long lGetListSize(String key) {  try {  return redisTemplate.opsForList().size(key);  } catch (Exception e) {  e.printStackTrace();  return 0;  }  }  /\*\*  \* 通过索引 获取list中的值  \*  \* @param key  \* 键  \* @param index  \* 索引 index>=0时， 0 表头，1 第二个元素，依次类推；index<0时，-1，表尾，-2倒数第二个元素，依次类推  \* @return  \*/  public Object lGetIndex(String key, long index) {  try {  return redisTemplate.opsForList().index(key, index);  } catch (Exception e) {  e.printStackTrace();  return null;  }  }  /\*\*  \* 将list放入缓存  \*  \* @param key  \* 键  \* @param value  \* 值  \* @param time  \* 时间(秒)  \* @return  \*/  public boolean lSet(String key, Object value) {  try {  redisTemplate.opsForList().rightPush(key, value);  return true;  } catch (Exception e) {  e.printStackTrace();  return false;  }  }  /\*\*  \* 将list放入缓存  \*  \* @param key  \* 键  \* @param value  \* 值  \* @param time  \* 时间(秒)  \* @return  \*/  public boolean lSet(String key, Object value, long time) {  try {  redisTemplate.opsForList().rightPush(key, value);  if (time > 0)  expire(key, time);  return true;  } catch (Exception e) {  e.printStackTrace();  return false;  }  }  /\*\*  \* 将list放入缓存  \*  \* @param key  \* 键  \* @param value  \* 值  \* @param time  \* 时间(秒)  \* @return  \*/  public boolean lSet(String key, List<Object> value) {  try {  redisTemplate.opsForList().rightPushAll(key, value);  return true;  } catch (Exception e) {  e.printStackTrace();  return false;  }  }  /\*\*  \* 将list放入缓存  \*  \* @param key  \* 键  \* @param value  \* 值  \* @param time  \* 时间(秒)  \* @return  \*/  public boolean lSet(String key, List<Object> value, long time) {  try {  redisTemplate.opsForList().rightPushAll(key, value);  if (time > 0)  expire(key, time);  return true;  } catch (Exception e) {  e.printStackTrace();  return false;  }  }  /\*\*  \* 根据索引修改list中的某条数据  \*  \* @param key  \* 键  \* @param index  \* 索引  \* @param value  \* 值  \* @return  \*/  public boolean lUpdateIndex(String key, long index, Object value) {  try {  redisTemplate.opsForList().set(key, index, value);  return true;  } catch (Exception e) {  e.printStackTrace();  return false;  }  }  /\*\*  \* 移除N个值为value  \*  \* @param key  \* 键  \* @param count  \* 移除多少个  \* @param value  \* 值  \* @return 移除的个数  \*/  public long lRemove(String key, long count, Object value) {  try {  Long remove = redisTemplate.opsForList().remove(key, count, value);  return remove;  } catch (Exception e) {  e.printStackTrace();  return 0;  }  }  } |

#### testRedis

|  |
| --- |
| package com.yaosiyuan.redis;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  public class TestRedis {  public static void main(String[] args) throws Exception {  @SuppressWarnings("resource")  ApplicationContext context = new ClassPathXmlApplicationContext("classpath:applicationContext-redis.xml");  RedisUtil redisUtil = (RedisUtil) context.getBean("redisUtil");  // =====================testString======================  redisUtil.set("name", "小高");  redisUtil.set("age", 26);  redisUtil.set("address", "河北邯郸");  // System.out.println(redisUtil.set("address", "河北邯郸", 50));  System.out.println(redisUtil.get("age"));  // redisUtil.set("age", 1000);  // Object object = redisUtil.get("user2");  // System.out.println(object);  // redisUtil.del("address");  // redisUtil.set("class", 15);  // long incr = redisUtil.incr("a", 1);  // System.out.println(incr);  // Thread.sleep(5000);  /\*  \* Map<String,Object> map=new HashMap<>(); map.put("name", "小白");  \* map.put("age", 24); map.put("address", "河北邯郸666");  \* redisUtil.hmset("15532002725", map,1000);  \*/  // redisUtil.del("15532002725");  // redisUtil.hset("15532002725","address","河北邯郸",1000);  // redisUtil.hdel("15532002725", "name");  // System.out.println(redisUtil.sSetAndTime("15532002727",1000,"haha"));  // System.out.println(redisUtil.sGet("15532002727"));  // System.out.println(redisUtil.sHasKey("15532002727","name"));  System.out.println(redisUtil.lRemove("15532002728", 1, 2));  System.out.println(redisUtil.lGet("15532002728", 0, -1));  System.out.println(redisUtil.lGetListSize("15532002728"));  System.out.println(redisUtil.lGetIndex("15532002728", 1));  // System.out.println(redisUtil.getExpire("15532002725"));  // System.out.println(redisUtil.hget("15532002725","name"));  // System.out.println(redisUtil.hmget("15532002725"));  }  }  class User {  private String name;  private Integer age;  private String address;  private Double classz;  private Float classz2;  public User() {  super();  }  public User(String name, Integer age, String address, Double classz, Float classz2) {  super();  this.name = name;  this.age = age;  this.address = address;  this.classz = classz;  this.classz2 = classz2;  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  public Integer getAge() {  return age;  }  public void setAge(Integer age) {  this.age = age;  }  public String getAddress() {  return address;  }  public void setAddress(String address) {  this.address = address;  }  public Double getClassz() {  return classz;  }  public void setClassz(Double classz) {  this.classz = classz;  }  public Float getClassz2() {  return classz2;  }  public void setClassz2(Float classz2) {  this.classz2 = classz2;  }  } |

### SSM整合redis

一、关于SSM整合Redis需要知道的一些小知识

在学习Redis的时候，大家应该知道，JAVA操作redis通常使用的是Jedis，通过java代码来操作redis的数据存储读取等操作，用过的人应该知道，Jedis客户端已经足够简单和轻量级了，但是呢，在此同时，Spring也为Redis提供了支持，就是在Spring-data模块中的Spring-Data-Redis（SDR），它一部分是基于Jedis客户端的API封装，另一部分是对Spring容器的整合。

大家应该都知道，Spring容器可以作为一个大工厂，为各种对象创建实例，关于SSM与Redis的整合，准确点，其实也就是Spring与Redis的整合。

又说会来了，Spring要和Redis整合需要用到的，就是Spring-Data模块中的Spring-data-Redis。

二、SDR资料(Spring-Data-Redis)

资料：http://projects.spring.io/spring-data-redis/

文档：https://docs.spring.io/spring-data/redis/docs/1.8.6.RELEASE/reference/html/

个人觉得没有什么比官方文档说的更为详细的了

这里对其功能做一个简单的说明：

1、连接池自动管理，并且提供了RedisTemplate类（很重要）

2、将jedis的API按照类型操作，进行封装

#### jar

添加jar包

在pom.xml文件中添加Redis所需的jar包

注意，本框架使用的是spring 4.3.14.RELEASE版本，redis 2.9.0版本的，spring-redis 1.8.10.RELEASE版本的，一定要看好版本，有的版本是不支持的。

另外本框架使用的是spring面向切面引入的Redis。

<!-- spring版本号 -->

<spring.version>4.3.14.RELEASE</spring.version>

<!-- mybatis版本号 -->

<mybatis.version>3.2.6</mybatis.version>

<!-- redis客户端jar -->

<dependency>

<groupId>redis.clients</groupId>

<artifactId>jedis</artifactId>

<version>2.9.0</version>

</dependency>

<!-- spring-redis实现 -->

<dependency>

<groupId>org.springframework.data</groupId>

<artifactId>spring-data-redis</artifactId>

<version>1.8.10.RELEASE</version>

</dependency>

<!-- spring 切面插件 start -->

<dependency>

<groupId>aopalliance</groupId>

<artifactId>aopalliance</artifactId>

<version>1.0</version>

</dependency>

<dependency>

<groupId>aspectj</groupId>

<artifactId>aspectjrt</artifactId>

<version>1.5.4</version>

</dependency>

<dependency>

<groupId>org.aspectj</groupId>

<artifactId>aspectjweaver</artifactId>

<version>1.9.2</version>

</dependency>

<!-- spring 切面插件 end-->

#### redis.properties

# Redis settings

redis.host=127.0.0.1

redis.port=6379

redis.pass=

redis.maxIdle=400

redis.maxTotal=6000

redis.maxWaitMillis=1000

redis.blockWhenExhausted=true

redis.testOnBorrow=true

redis.timeout=100000

defaultCacheExpireTime=3600

#### spring-redis.xml

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

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:p="http://www.springframework.org/schema/p"

xmlns:context="http://www.springframework.org/schema/context"

xmlns:mvc="http://www.springframework.org/schema/mvc" xmlns:aop="http://www.springframework.org/schema/aop"

xmlns:tx="http://www.springframework.org/schema/tx"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-4.0.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-4.0.xsd

http://www.springframework.org/schema/mvc

http://www.springframework.org/schema/mvc/spring-mvc-4.0.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop-4.0.xsd

http://www.springframework.org/schema/tx

http://www.springframework.org/schema/tx/spring-tx.xsd">

<!-- redis数据源 -->

<bean id="poolConfig" class="redis.clients.jedis.JedisPoolConfig">

<!-- 最大空闲数 -->

<property name="maxIdle" value="${redis.maxIdle}" />

<!-- 最大空连接数 -->

<property name="maxTotal" value="${redis.maxTotal}" />

<!-- 最大等待时间 -->

<property name="maxWaitMillis" value="${redis.maxWaitMillis}" />

<!-- 连接超时时是否阻塞，false时报异常,ture阻塞直到超时, 默认true -->

<property name="blockWhenExhausted" value="${redis.blockWhenExhausted}" />

<!-- 返回连接时，检测连接是否成功 -->

<property name="testOnBorrow" value="${redis.testOnBorrow}" />

</bean>

<!-- Spring-redis连接池管理工厂 -->

<bean id="jedisConnectionFactory"

class="org.springframework.data.redis.connection.jedis.JedisConnectionFactory"

p:hostName="${redis.host}" p:port="${redis.port}" p:password="${redis.pass}"

p:timeout="${redis.timeout}" p:poolConfig-ref="poolConfig" p:usePool="true">

</bean>

<!-- redis template definition -->

<bean id="redisTemplate" class="org.springframework.data.redis.core.RedisTemplate">

<property name="connectionFactory" ref="jedisConnectionFactory" />

<property name="keySerializer">

<bean

class="org.springframework.data.redis.serializer.StringRedisSerializer" />

</property>

<property name="valueSerializer">

<bean

class="org.springframework.data.redis.serializer.JdkSerializationRedisSerializer" />

</property>

<property name="hashKeySerializer">

<bean

class="org.springframework.data.redis.serializer.StringRedisSerializer" />

</property>

<property name="hashValueSerializer">

<bean

class="org.springframework.data.redis.serializer.JdkSerializationRedisSerializer" />

</property>

<!--开启事务 -->

<property name="enableTransactionSupport" value="true"></property>

</bean>

<!-- 缓存拦截器配置 -->

<bean id="methodCacheInterceptor" class="com.ivan.redis.MethodCacheInterceptor">

<property name="redisUtil" ref="redisUtil" />

<property name="defaultCacheExpireTime" value="${defaultCacheExpireTime}" />

<!-- 禁用缓存的类名列表 -->

<property name="targetNamesList">

<list>

<value></value>

</list>

</property>

<!-- 禁用缓存的方法名列表 -->

<property name="methodNamesList">

<list>

<value>add</value>

<value>delete</value>

<value>edit</value>

</list>

</property>

</bean>

<bean id="redisUtil" class="com.ivan.redis.RedisUtil">

<property name="redisTemplate" ref="redisTemplate" />

</bean>

<!--配置切面拦截方法 -->

<aop:config proxy-target-class="true">

<!-- <aop:pointcut id="controllerMethodPointcut"

expression="execution(\* com.ivan.service.impl.\*.get\*(..))" /> -->

<aop:pointcut id="controllerMethodPointcut"

expression="execution(\* com.ivan.service.impl.\*.\*(..))" />

<aop:advisor advice-ref="methodCacheInterceptor"

pointcut-ref="controllerMethodPointcut" />

</aop:config>

</beans>

### mybatis二级缓存Redis

https://blog.csdn.net/zengxiantao1994/article/details/79759728

Redis是一个开源的使用ANSI C语言编写、支持网络、可基于内存亦可持久化的日志型、Key-Value数据库，并提供多种语言的API。

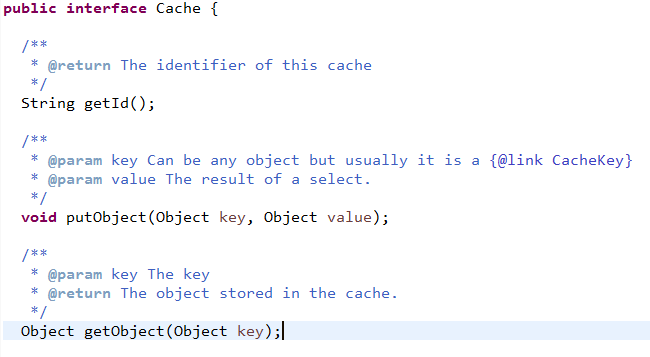
        不使用分布式缓存，缓存数据在各自的服务器中存储，不方便系统开发，所以要使用分布式缓存对缓存数据进行集中式管理。

        mybatis本身无法实现分布式缓存，需要和其他的分布式缓存进行整合。整合方法：mybatis提供了一个Cache接口，如果要实现自己的缓存逻辑，只需要实现Cache接口即可。

#### Mybatis默认缓存机制

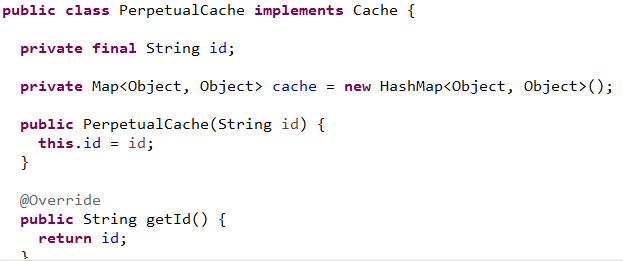
mybatis提供二级缓存Cache接口，如下：

https://img-blog.csdn.net/20180330163712688



它的默认实现类：

https://img-blog.csdn.net/20180330163750559



通过实现Cache接口可以实现mybatis缓存数据通过其它缓存数据库整合，mybatis的特长是sql操作，缓存数据的管理不是mybatis的特长，为了提高缓存的性能可以将mybatis和第三方的缓存数据库整合，比如ehcache、memcache、redis等。

#### Mybatis与Redis的整合

首先需要配置好SSM的开发环境，然后：

1、引入jar包

https://img-blog.csdn.net/20180330163819240https://img-blog.csdn.net/20180330163831155

2、在Mybatis的配置文件中开启缓存设置

二、mybatis集成redis进行缓存配置

1.mybatis开启缓存支持

在spring-mabatis.xml中添加下列内容：

<!-- spring和MyBatis完美整合，不需要mybatis的配置映射文件 -->

<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">

<property name="dataSource" ref="dataSource" />

<!-- 自动扫描mapping.xml文件 -->

<property name="mapperLocations" value="classpath:com/dg/mapping/\*.xml"></property>

<!-- spring和MyBatis完美整合，不需要mybatis的配置映射文件 -->

<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">

<property name="dataSource" ref="dataSource" />

<!-- 自动扫描mapping.xml文件 -->

<property name="mapperLocations" value="classpath:com/yaosiyuan/mapping/\*.xml"></property>

<!-- 开启缓存支持 -->

<property name="configurationProperties">

<props>

<prop key="cacheEnabled">true</prop>

<!-- 查询时，关闭关联对象即时加载以提高性能 -->

<prop key="lazyLoadingEnabled">false</prop>

<!-- 设置关联对象加载的形态，此处为按需加载字段(加载字段由SQL指定)，不会加载关联表的所有字段，以提高性能 -->

<prop key="aggressiveLazyLoading">true</prop>

<!-- 对于未知的SQL查询，允许返回不同的结果集以达到通用的效果 -->

<prop key="multipleResultSetsEnabled">true</prop>

<!-- 允许使用列标签代替列名 -->

<prop key="useColumnLabel">true</prop>

<!-- 允许使用自定义的主键值(比如由程序生成的UUID 32位编码作为键值)，数据表的PK生成策略将被覆盖 -->

<prop key="useGeneratedKeys">true</prop>

<!-- 给予被嵌套的resultMap以字段-属性的映射支持 -->

<prop key="autoMappingBehavior">FULL</prop>

<!-- 对于批量更新操作缓存SQL以提高性能 -->

<prop key="defaultExecutorType">BATCH</prop>

<!-- 数据库超过25000秒仍未响应则超时 -->

<prop key="defaultStatementTimeout">25000</prop>

</props>

</property>

</bean>

</bean>

#### MyBatis内置的二级缓存算法

Mybatis的所有Cache算法都是基于装饰器模式对PerpetualCache扩展增加功能。

1）FIFO:先入先出,基于LinkedList实现；

2）LRU:最近最少使用，基于LinkedHashMap实现，在put的时候，自动移除最少使用缓存对象；

3）SOFT:对Cache的value进行SoftReference包装；当缓存对象是Soft reference可达时，gc会向系统申请更多内存，而不是直接回收它，当内存不足的时候才回收它；

4）WEAK:对Cache的value进行WeakReference包装；WeakReference不会强制对象保存在内存中。它拥有比较短暂的生命周期，允许你使用垃圾回收器的能力去权衡一个对象的可达性。在垃圾回收器扫描它所管辖的内存区域过程中，一旦gc发现对象是weakReference可达，就会把它放到ReferenceQueue中，等下次gc时回收它。

## 工具类

### JsonUtil

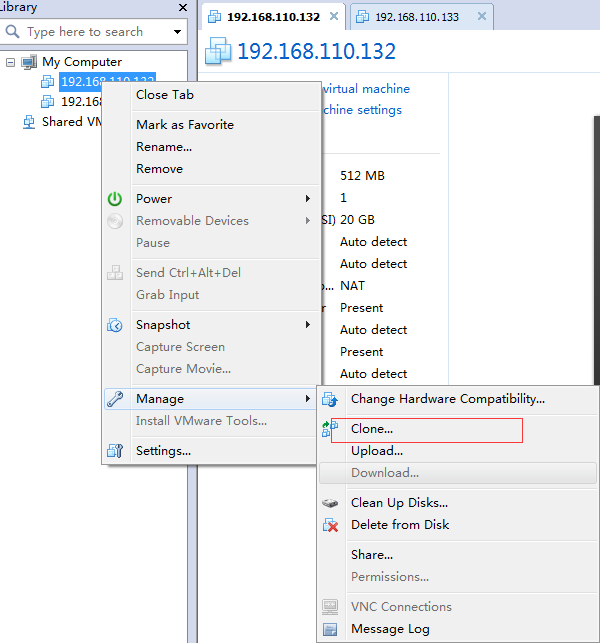
|  |
| --- |
| package com.itheima.utils;  import java.util.List;  import java.util.Map;  import net.sf.json.JSONArray;  import net.sf.json.JSONObject;  import net.sf.json.JsonConfig;  import net.sf.json.util.CycleDetectionStrategy;  import net.sf.json.xml.XMLSerializer;  /\*\*  \* 处理json数据格式的工具类  \*  \* @Date 2013-3-31  \* @version 1.0  \*/  public class JsonUtil {  /\*\*  \* 将数组转换成String类型的JSON数据格式  \*  \* @param objects  \* @return  \*/  public static String array2json(Object[] objects){    JSONArray jsonArray = JSONArray.fromObject(objects);  return jsonArray.toString();    }    /\*\*  \* 将list集合转换成String类型的JSON数据格式  \*  \* @param list  \* @return  \*/  public static String list2json(List list){    JSONArray jsonArray = JSONArray.fromObject(list);  return jsonArray.toString();    }    /\*\*  \* 将map集合转换成String类型的JSON数据格式  \*  \* @param map  \* @return  \*/  public static String map2json(Map map){    JSONObject jsonObject = JSONObject.fromObject(map);  return jsonObject.toString();    }    /\*\*  \* 将Object对象转换成String类型的JSON数据格式  \*  \* @param object  \* @return  \*/  public static String object2json(Object object){    JSONObject jsonObject = JSONObject.fromObject(object);  return jsonObject.toString();    }    /\*\*  \* 将XML数据格式转换成String类型的JSON数据格式  \*  \* @param xml  \* @return  \*/  public static String xml2json(String xml){    JSONArray jsonArray = (JSONArray) new XMLSerializer().read(xml);  return jsonArray.toString();    }    /\*\*  \* 除去不想生成的字段（特别适合去掉级联的对象）  \*  \* @param excludes  \* @return  \*/  public static JsonConfig configJson(String[] excludes) {  JsonConfig jsonConfig = new JsonConfig();  jsonConfig.setExcludes(excludes);  jsonConfig.setIgnoreDefaultExcludes(true);  jsonConfig.setCycleDetectionStrategy(CycleDetectionStrategy.LENIENT);  return jsonConfig;  }    } |

## 高级

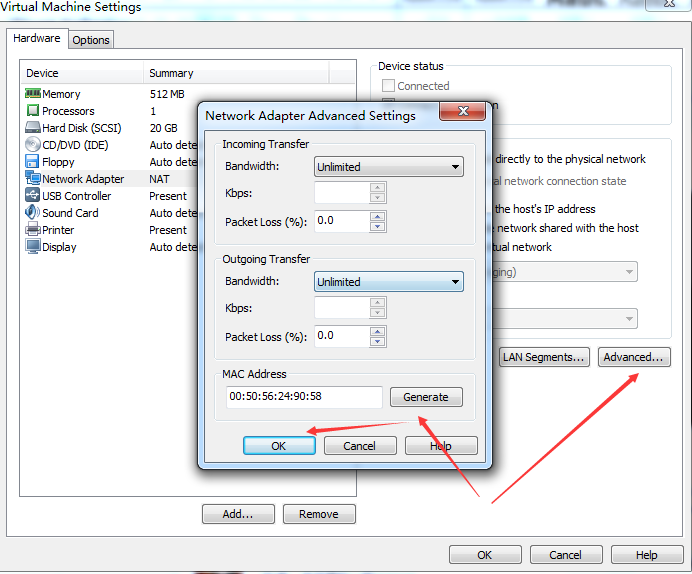
### Redis主从复制

克隆三台linux虚拟机

#### 克隆虚拟机



#### 生成新的mack地址



#### 主从复制配置

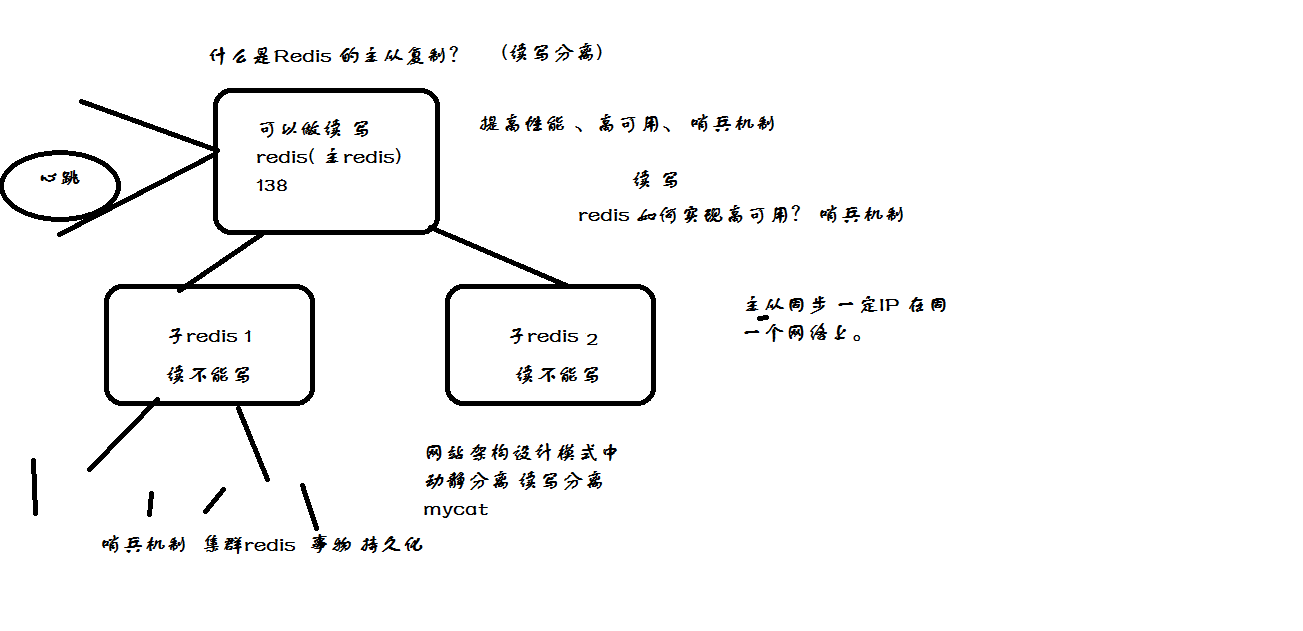
redis主从复制

概述

1、redis的复制功能是支持多个数据库之间的数据同步。一类是主数据库（master）一类是从数据库（slave），主数据库可以进行读写操作，当发生写操作的时候自动将数据同步到从数据库，而从数据库一般是只读的，并接收主数据库同步过来的数据，一个主数据库可以有多个从数据库，而一个从数据库只能有一个主数据库。

2、通过redis的复制功能可以很好的实现数据库的读写分离，提高服务器的负载能力。主数据库主要进行写操作，而从数据库负责读操作。

主从复制过程



主从复制过程：见下图



**过程：**

1：当一个从数据库启动时，会向主数据库发送sync命令，

2：主数据库接收到sync命令后会开始在后台保存快照（执行rdb操作），并将保存期间接收到的命令缓存起来

3：当快照完成后，redis会将快照文件和所有缓存的命令发送给从数据库。

4：从数据库收到后，会载入快照文件并执行收到的缓存的命令。

##### 修改redis.conf

修改从redis中的 redis.conf文件

slaveof 192.168.33.130 6379

masterauth 123456--- 主redis服务器配置了密码,则需要配置

### Redis哨兵机制

#### 什么是哨兵机制

Redis的哨兵(sentinel) 系统用于管理多个 Redis 服务器,该系统执行以下三个任务:

· **监控(Monitoring)**: 哨兵(sentinel) 会不断地检查你的Master和Slave是否运作正常。

· **提醒(Notification)**:当被监控的某个 Redis出现问题时, 哨兵(sentinel) 可以通过 API 向管理员或者其他应用程序发送通知。

· **自动故障迁移(Automatic failover)**:当一个Master不能正常工作时，哨兵(sentinel) 会开始一次自动故障迁移操作,它会将失效Master的其中一个Slave升级为新的Master, 并让失效Master的其他Slave改为复制新的Master; 当客户端试图连接失效的Master时,集群也会向客户端返回新Master的地址,使得集群可以使用Master代替失效Master。

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.

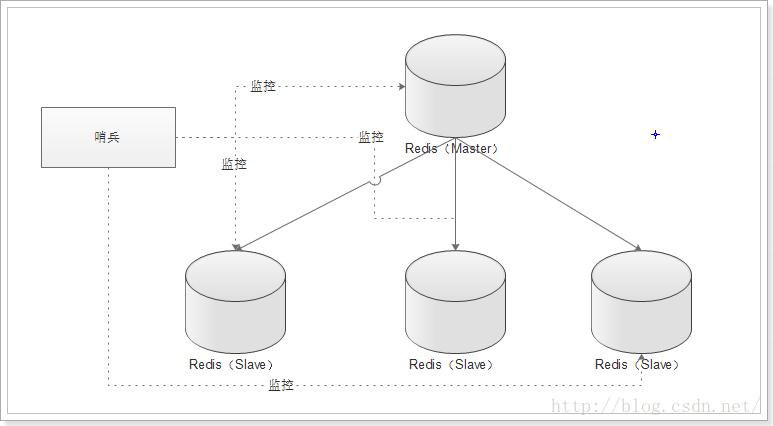
每个哨兵(sentinel) 会向其它哨兵(sentinel)、master、slave定时发送消息,以确认对方是否”活”着,如果发现对方在指定时间(可配置)内未回应,则暂时认为对方已挂(**所谓的”主观认为宕机” Subjective Down,简称sdown**).

若“哨兵群”中的多数sentinel,都报告某一master没响应,系统才认为该master"彻底死亡"(**即:客观上的真正down机,Objective Down,简称odown**),通过一定的vote算法,从剩下的slave节点中,选一台提升为master,然后自动修改相关配置.

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

哨兵(sentinel) 的一些设计思路和zookeeper非常类似

单个哨兵(sentinel)



#### 哨兵模式修改配置

info 命令查看 当前信息

实现步骤:

1.拷贝到etc目录

cp sentinel.conf /usr/local/redis/etc

2.修改sentinel.conf配置文件

sentinel monitor mymast 192.168.110.133 6379 1 #主节点 名称 IP 端口号 选举次数

3. 修改心跳检测 5000毫秒

sentinel down-after-milliseconds mymaster 5000

4.sentinel parallel-syncs mymaster 2 --- 最多有多少合格节点

关闭redis redis-cli shutdown

5. 启动哨兵模式

./redis-server /usr/local/redis/etc/sentinel.conf --sentinel &

启动哨兵模式 还需要启动redis

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

6. 停止哨兵模式

### Redis事物

#### Redis事物

Redis 事务可以一次执行多个命令， 并且带有以下两个重要的保证：

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

事务是一个原子操作：事务中的命令要么全部被执行，要么全部都不执行。

一个事务从开始到执行会经历以下三个阶段：

开始事务。

命令入队。

执行事务。

#### 实例

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

|  |
| --- |
| redis 127.0.0.1:6379> MULTI  OK  redis 127.0.0.1:6379> SET book-name "Mastering C++ in 21 days"  QUEUED  redis 127.0.0.1:6379> GET book-name  QUEUED  redis 127.0.0.1:6379> SADD tag "C++" "Programming" "Mastering Series"  QUEUED  redis 127.0.0.1:6379> SMEMBERS tag  QUEUED  redis 127.0.0.1:6379> EXEC  1) OK  2) "Mastering C++ in 21 days"  3) (integer) 3  4) 1) "Mastering Series"  2) "C++"  3) "Programming" |

#### Redis 事务命令

下表列出了 redis 事务的相关命令：

|  |  |
| --- | --- |
| **序号** | **命令及描述** |
| 1 | [DISCARD](http://www.runoob.com/redis/transactions-discard.html) 取消事务，放弃执行事务块内的所有命令。 |
| 2 | [EXEC](http://www.runoob.com/redis/transactions-exec.html) 执行所有事务块内的命令。 |
| 3 | [MULTI](http://www.runoob.com/redis/transactions-multi.html) 标记一个事务块的开始。 |
| 4 | [UNWATCH](http://www.runoob.com/redis/transactions-unwatch.html) 取消 WATCH 命令对所有 key 的监视。 |
| 5 | [WATCH key [key ...]](http://www.runoob.com/redis/transactions-watch.html) 监视一个(或多个) key ，如果在事务执行之前这个(或这些) key 被其他命令所改动，那么事务将被打断。 |

### Redis持久化

##### 什么是Redis持久化

什么是Redis持久化,就是将内存数据保存到硬盘。

Redis 持久化存储 (AOF 与 RDB 两种模式)

##### RDB持久化

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**

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

RDB 默认开启，redis.conf 中的具体配置参数如下；

|  |
| --- |
| #dbfilename：持久化数据存储在本地的文件  dbfilename dump.rdb  #dir：持久化数据存储在本地的路径，如果是在/redis/redis-3.0.6/src下启动的redis-cli，则数据会存储在当前src目录下  dir ./  ##snapshot触发的时机，save  ##如下为900秒后，至少有一个变更操作，才会snapshot  ##对于此值的设置，需要谨慎，评估系统的变更操作密集程度  ##可以通过“save “””来关闭snapshot功能  #save时间，以下分别表示更改了1个key时间隔900s进行持久化存储；更改了10个key300s进行存储；更改10000个key60s进行存储。  save 900 1  save 300 10  save 60 10000  ##当snapshot时出现错误无法继续时，是否阻塞客户端“变更操作”，“错误”可能因为磁盘已满/磁盘故障/OS级别异常等  stop-writes-on-bgsave-error yes  ##是否启用rdb文件压缩，默认为“yes”，压缩往往意味着“额外的cpu消耗”，同时也意味这较小的文件尺寸以及较短的网络传输时间  rdbcompression yes |

##### AOF持久化

Append-only file，将“操作 + 数据”以格式化指令的方式追加到操作日志文件的尾部，在 append 操作返回后(已经写入到文件或者即将写入)，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。

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

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

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

先将rdp模式关闭



|  |
| --- |
| ##此选项为aof功能的开关，默认为“no”，可以通过“yes”来开启aof功能  ##只有在“yes”下，aof重写/文件同步等特性才会生效  appendonly yes  ##指定aof文件名称  appendfilename appendonly.aof  ##指定aof操作中文件同步策略，有三个合法值：always everysec no,默认为everysec  appendfsync everysec  ##在aof-rewrite期间，appendfsync是否暂缓文件同步，"no"表示“不暂缓”，“yes”表示“暂缓”，默认为“no”  no-appendfsync-on-rewrite no  ##aof文件rewrite触发的最小文件尺寸(mb,gb),只有大于此aof文件大于此尺寸是才会触发rewrite，默认“64mb”，建议“512mb”  auto-aof-rewrite-min-size 64mb  ##相对于“上一次”rewrite，本次rewrite触发时aof文件应该增长的百分比。  ##每一次rewrite之后，redis都会记录下此时“新aof”文件的大小(例如A)，那么当aof文件增长到A\*(1 + p)之后  ##触发下一次rewrite，每一次aof记录的添加，都会检测当前aof文件的尺寸。  auto-aof-rewrite-percentage 100 |

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

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

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

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

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

AOF 文件会不断增大，它的大小直接影响“故障恢复”的时间, 而且 AOF 文件中历史操作是可以丢弃的。AOF rewrite 操作就是“压缩”AOF 文件的过程，当然 redis 并没有采用“基于原 aof 文件”来重写的方式，而是采取了类似 snapshot 的方式：基于 copy-on-write，全量遍历内存中数据，然后逐个序列到 aof 文件中。因此 AOF rewrite 能够正确反应当前内存数据的状态，这正是我们所需要的；\*rewrite 过程中，对于新的变更操作将仍然被写入到原 AOF 文件中，同时这些新的变更操作也会被 redis 收集起来(buffer，copy-on-write 方式下，最极端的可能是所有的 key 都在此期间被修改，将会耗费 2 倍内存)，当内存数据被全部写入到新的 aof 文件之后，收集的新的变更操作也将会一并追加到新的 aof 文件中，此后将会重命名新的 aof 文件为 appendonly.aof, 此后所有的操作都将被写入新的 aof 文件。如果在 rewrite 过程中，出现故障，将不会影响原 AOF 文件的正常工作，只有当 rewrite 完成之后才会切换文件，因为 rewrite 过程是比较可靠的。\*

触发 rewrite 的时机可以通过配置文件来声明，同时 redis 中可以通过 bgrewriteaof 指令人工干预。

redis-cli -h ip -p port bgrewriteaof

因为 rewrite 操作 /aof 记录同步 /snapshot 都消耗磁盘 IO，redis 采取了“schedule”策略：无论是“人工干预”还是系统触发，snapshot 和 rewrite 需要逐个被执行。

AOF rewrite 过程并不阻塞客户端请求。系统会开启一个子进程来完成。

##### AOF与RDB区别

OF 和 RDB 各有优缺点，这是有它们各自的特点所决定：

1) AOF 更加安全，可以将数据更加及时的同步到文件中，但是 AOF 需要较多的磁盘 IO 开支，AOF 文件尺寸较大，文件内容恢复数度相对较慢。

\*2) snapshot，安全性较差，它是“正常时期”数据备份以及 master-slave 数据同步的最佳手段，文件尺寸较小，恢复数度较快。

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

### Redis发布订阅

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

Redis 客户端可以订阅任意数量的频道。

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



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



##### 13.1实例

以下实例演示了发布订阅是如何工作的。在我们实例中我们创建了订阅频道名为 **redisChat**:

redis 127.0.0.1:6379> SUBSCRIBE redisChat

Reading messages... (press Ctrl-C to quit)

1) "subscribe"

2) "redisChat"

3) (integer) 1

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

redis 127.0.0.1:6379> PUBLISH redisChat "Redis is a great caching technique"

(integer) 1

redis 127.0.0.1:6379> PUBLISH redisChat "Learn redis by runoob.com"

(integer) 1

# 订阅者的客户端会显示如下消息

1) "message"

2) "redisChat"

3) "Redis is a great caching technique"

1) "message"

2) "redisChat"

3) "Learn redis by runoob.com"

#### Redis 发布订阅命令

下表列出了 redis 发布订阅常用命令：

|  |  |
| --- | --- |
| **序号** | **命令及描述** |
| 1 | [PSUBSCRIBE pattern [pattern ...]](http://www.runoob.com/redis/pub-sub-psubscribe.html) 订阅一个或多个符合给定模式的频道。 |
| 2 | [PUBSUB subcommand [argument [argument ...]]](http://www.runoob.com/redis/pub-sub-pubsub.html) 查看订阅与发布系统状态。 |
| 3 | [PUBLISH channel message](http://www.runoob.com/redis/pub-sub-publish.html) 将信息发送到指定的频道。 |
| 4 | [PUNSUBSCRIBE [pattern [pattern ...]]](http://www.runoob.com/redis/pub-sub-punsubscribe.html) 退订所有给定模式的频道。 |
| 5 | [SUBSCRIBE channel [channel ...]](http://www.runoob.com/redis/pub-sub-subscribe.html) 订阅给定的一个或多个频道的信息。 |
| 6 | [UNSUBSCRIBE [channel [channel ...]]](http://www.runoob.com/redis/pub-sub-unsubscribe.html) 指退订给定的频道。 |

### Redis集群

#### 常见问题

异常信息

[ERR] Node 192.168.25.130:7001 is not empty. Either the node already knows other nodes (check with CLUSTER NODES) or contains some key in database 0.

Can I set the above configuration? (type 'yes' to accept): yes

/usr/lib/ruby/gems/1.8/gems/redis-3.0.0/lib/redis/client.rb:79:in `call': ERR Slot 4648 is already busy (Redis::CommandError)

from /usr/lib/ruby/gems/1.8/gems/redis-3.0.0/lib/redis.rb:2190:in `method\_missing'

from /usr/lib/ruby/gems/1.8/gems/redis-3.0.0/lib/redis.rb:36:in `synchronize'

from /usr/lib/ruby/1.8/monitor.rb:242:in `mon\_synchronize'

from /usr/lib/ruby/gems/1.8/gems/redis-3.0.0/lib/redis.rb:36:in `synchronize'

from /usr/lib/ruby/gems/1.8/gems/redis-3.0.0/lib/redis.rb:2189:in `method\_missing'

from ./redis-trib.rb:205:in `flush\_node\_config'

from ./redis-trib.rb:657:in `flush\_nodes\_config'

from ./redis-trib.rb:656:in `each'

from ./redis-trib.rb:656:in `flush\_nodes\_config'

from ./redis-trib.rb:997:in `create\_cluster\_cmd'

from ./redis-trib.rb:1373:in `send'

from ./redis-trib.rb:1373

场景描述

后端代码报错，连接不上redis集群。然后重启，重新创建集群报错。

后端报错代码

二月 21, 2019 7:47:13 下午 org.apache.catalina.core.StandardWrapperValve invoke

SEVERE: Servlet.service() for servlet [e3-sso-web] in context with path [] threw exception [Request processing failed; nested exception is java.lang.RuntimeException: redis.clients.jedis.exceptions.JedisClusterException: CLUSTERDOWN The cluster is down

redis.clients.jedis.exceptions.JedisClusterException: CLUSTERDOWN The cluster is down

at redis.clients.jedis.Protocol.processError(Protocol.java:115)

at redis.clients.jedis.Protocol.process(Protocol.java:142)

at redis.clients.jedis.Protocol.read(Protocol.java:196)

at redis.clients.jedis.Connection.readProtocolWithCheckingBroken(Connection.java:288)

at redis.clients.jedis.Connection.getStatusCodeReply(Connection.java:187)

at redis.clients.jedis.Jedis.set(Jedis.java:66)

at redis.clients.jedis.JedisCluster$1.execute(JedisCluster.java:73)

at redis.clients.jedis.JedisCluster$1.execute(JedisCluster.java:70)

at redis.clients.jedis.JedisClusterCommand.runWithRetries(JedisClusterCommand.java:56)

at redis.clients.jedis.JedisClusterCommand.runWithRetries(JedisClusterCommand.java:67)

at redis.clients.jedis.JedisClusterCommand.run(JedisClusterCommand.java:29)

at redis.clients.jedis.JedisCluster.set(JedisCluster.java:70)

at cn.e3mall.common.jedis.JedisClientCluster.set(JedisClientCluster.java:22)

at cn.e3mall.sso.service.impl.LoginServiceImpl.userLogin(LoginServiceImpl.java:65)

at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)

at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)

at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)

at java.lang.reflect.Method.invoke(Method.java:601)

at org.springframework.aop.support.AopUtils.invokeJoinpointUsingReflection(AopUtils.java:302)

at org.springframework.aop.framework.ReflectiveMethodInvocation.invokeJoinpoint(ReflectiveMethodInvocation.java:190)

at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMethodInvocation.java:157)

at org.springframework.transaction.interceptor.TransactionInterceptor$1.proceedWithInvocation(TransactionInterceptor.java:99)

at org.springframework.transaction.interceptor.TransactionAspectSupport.invokeWithinTransaction(TransactionAspectSupport.java:281)

at org.springframework.transaction.interceptor.TransactionInterceptor.invoke(TransactionInterceptor.java:96)

at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMethodInvocation.java:179)

at org.springframework.aop.interceptor.ExposeInvocationInterceptor.invoke(ExposeInvocationInterceptor.java:92)

at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMethodInvocation.java:179)

at org.springframework.aop.framework.JdkDynamicAopProxy.invoke(JdkDynamicAopProxy.java:208)

at $Proxy34.userLogin(Unknown Source)

at com.alibaba.dubbo.common.bytecode.Wrapper1.invokeMethod(Wrapper1.java)

at com.alibaba.dubbo.rpc.proxy.javassist.JavassistProxyFactory$1.doInvoke(JavassistProxyFactory.java:46)

at com.alibaba.dubbo.rpc.proxy.AbstractProxyInvoker.invoke(AbstractProxyInvoker.java:72)

at com.alibaba.dubbo.rpc.protocol.InvokerWrapper.invoke(InvokerWrapper.java:53)

at com.alibaba.dubbo.rpc.filter.ExceptionFilter.invoke(ExceptionFilter.java:64)

at com.alibaba.dubbo.rpc.protocol.ProtocolFilterWrapper$1.invoke(ProtocolFilterWrapper.java:91)

at com.alibaba.dubbo.rpc.filter.TimeoutFilter.invoke(TimeoutFilter.java:42)

at com.alibaba.dubbo.rpc.protocol.ProtocolFilterWrapper$1.invoke(ProtocolFilterWrapper.java:91)

at com.alibaba.dubbo.monitor.support.MonitorFilter.invoke(MonitorFilter.java:75)

at com.alibaba.dubbo.rpc.protocol.ProtocolFilterWrapper$1.invoke(ProtocolFilterWrapper.java:91)

at com.alibaba.dubbo.rpc.protocol.dubbo.filter.TraceFilter.invoke(TraceFilter.java:78)

at com.alibaba.dubbo.rpc.protocol.ProtocolFilterWrapper$1.invoke(ProtocolFilterWrapper.java:91)

at com.alibaba.dubbo.rpc.filter.ContextFilter.invoke(ContextFilter.java:60)

at com.alibaba.dubbo.rpc.protocol.ProtocolFilterWrapper$1.invoke(ProtocolFilterWrapper.java:91)

at com.alibaba.dubbo.rpc.filter.GenericFilter.invoke(GenericFilter.java:112)

at com.alibaba.dubbo.rpc.protocol.ProtocolFilterWrapper$1.invoke(ProtocolFilterWrapper.java:91)

at com.alibaba.dubbo.rpc.filter.ClassLoaderFilter.invoke(ClassLoaderFilter.java:38)

at com.alibaba.dubbo.rpc.protocol.ProtocolFilterWrapper$1.invoke(ProtocolFilterWrapper.java:91)

at com.alibaba.dubbo.rpc.filter.EchoFilter.invoke(EchoFilter.java:38)

at com.alibaba.dubbo.rpc.protocol.ProtocolFilterWrapper$1.invoke(ProtocolFilterWrapper.java:91)

at com.alibaba.dubbo.rpc.protocol.dubbo.DubboProtocol$1.reply(DubboProtocol.java:108)

at com.alibaba.dubbo.remoting.exchange.support.header.HeaderExchangeHandler.handleRequest(HeaderExchangeHandler.java:84)

at com.alibaba.dubbo.remoting.exchange.support.header.HeaderExchangeHandler.received(HeaderExchangeHandler.java:170)

at com.alibaba.dubbo.remoting.transport.DecodeHandler.received(DecodeHandler.java:52)

at com.alibaba.dubbo.remoting.transport.dispatcher.ChannelEventRunnable.run(ChannelEventRunnable.java:82)

at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1110)

at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:603)

at java.lang.Thread.run(Thread.java:722)

] with root cause

java.lang.RuntimeException: redis.clients.jedis.exceptions.JedisClusterException: CLUSTERDOWN The cluster is down

redis.clients.jedis.exceptions.JedisClusterException: CLUSTERDOWN The cluster is down

异常原因

创建新集群需要没有缓存。是新环境。之前用过或者保存过信息，再次创建集群就会报错。



解决方案

删除每个redis节点下的dump.rdb和nodes.conf。

|  |
| --- |
| cd redis01  rm -rf dump.rdb  rm -rf nodes.conf  cd ..  cd redis02  rm -rf dump.rdb  rm -rf nodes.conf  cd ..  cd redis03  rm -rf dump.rdb  rm -rf nodes.conf  cd ..  cd redis04  rm -rf dump.rdb  rm -rf nodes.conf  cd ..  cd redis05  rm -rf dump.rdb  rm -rf nodes.conf  cd ..  cd redis06  rm -rf dump.rdb  rm -rf nodes.conf  cd .. |

登录每个redis节点，使用以下命令

Redis-cli连接集群。

[root@localhost redis-cluster]# redis01/redis-cli -p 7002 -c

-c：代表连接的是redis集群

flushdb

cluster reset

chmod u+x delete-all-rdb.sh

参考资料

https://blog.csdn.net/zhangxiaomo\_java/article/details/71189293

#### ERR Unsupported CONFIG parameter: notify-keyspace-events

**异常信息**

org.springframework.data.redis.RedisConnectionFailureException: Cannot get Jedis connection; nested exception is redis.clients.jedis.exceptions.JedisConnectionException: Could not get a resource from the pool at org.springframework.data.redis.connection.jedis.JedisConnectionFactory.fetchJedisConnector(JedisConnectionFactory.java:204) ~[spring-data-redis-1.8.18.RELEASE.jar:na] at org.springframework.data.redis.connection.jedis.JedisConnectionFactory.getConnection(JedisConnectionFactory.java:348) ~[spring-data-redis-1.8.18.RELEASE.jar:na] at org.springframework.data.redis.core.RedisConnectionUtils.doGetConnection(RedisConnectionUtils.java:129) ~[spring-data-redis-1.8.18.RELEASE.jar:na] at org.springframework.data.redis.core.RedisConnectionUtils.getConnection(RedisConnectionUtils.java:92) ~[spring-data-redis-1.8.18.RELEASE.jar:na] at org.springframework.data.redis.core.RedisConnectionUtils.getConnection(RedisConnectionUtils.java:79) ~[spring-data-redis-1.8.18.RELEASE.jar:na] at org.springframework.data.redis.core.RedisTemplate.execute(RedisTemplate.java:194) ~[spring-data-redis-1.8.18.RELEASE.jar:na] at org.springframework.data.redis.core.RedisTemplate.execute(RedisTemplate.java:169) ~[spring-data-redis-1.8.18.RELEASE.jar:na] at org.springframework.data.redis.core.AbstractOperations.execute(AbstractOperations.java:91) ~[spring-data-redis-1.8.18.RELEASE.jar:na] at org.springframework.data.redis.core.DefaultSetOperations.members(DefaultSetOperations.java:126) ~[spring-data-redis-1.8.18.RELEASE.jar:na] at org.springframework.data.redis.core.DefaultBoundSetOperations.members(DefaultBoundSetOperations.java:91) ~[spring-data-redis-1.8.18.RELEASE.jar:na] at org.springframework.session.data.redis.RedisSessionExpirationPolicy.cleanExpiredSessions(RedisSessionExpirationPolicy.java:131) ~[spring-session-1.3.5.RELEASE.jar:na] at org.springframework.session.data.redis.RedisOperationsSessionRepository.cleanupExpiredSessions(RedisOperationsSessionRepository.java:398) ~[spring-session-1.3.5.RELEASE.jar:na] at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method) ~[na:1.8.0\_144] at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62) ~[na:1.8.0\_144] at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43) ~[na:1.8.0\_144] at java.lang.reflect.Method.invoke(Method.java:498) ~[na:1.8.0\_144] at org.springframework.scheduling.support.ScheduledMethodRunnable.run(ScheduledMethodRunnable.java:65) ~[spring-context-4.3.22.RELEASE.jar:4.3.22.RELEASE] at org.springframework.scheduling.support.DelegatingErrorHandlingRunnable.run(DelegatingErrorHandlingRunnable.java:54) ~[spring-context-4.3.22.RELEASE.jar:4.3.22.RELEASE] at org.springframework.scheduling.concurrent.ReschedulingRunnable.run(ReschedulingRunnable.java:81) [spring-context-4.3.22.RELEASE.jar:4.3.22.RELEASE] at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:511) [na:1.8.0\_144] at java.util.concurrent.FutureTask.run(FutureTask.java:266) [na:1.8.0\_144] at java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask.access$201(ScheduledThreadPoolExecutor.java:180) [na:1.8.0\_144] at java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask.run(ScheduledThreadPoolExecutor.java:293) [na:1.8.0\_144] at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149) [na:1.8.0\_144] at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624) [na:1.8.0\_144] at java.lang.Thread.run(Thread.java:748) [na:1.8.0\_144] Caused by: redis.clients.jedis.exceptions.JedisConnectionException: Could not get a resource from the pool at redis.clients.util.Pool.getResource(Pool.java:53) ~[jedis-2.9.1.jar:na] at redis.clients.jedis.JedisPool.getResource(JedisPool.java:226) ~[jedis-2.9.1.jar:na] at redis.clients.jedis.JedisPool.getResource(JedisPool.java:16) ~[jedis-2.9.1.jar:na] at org.springframework.data.redis.connection.jedis.JedisConnectionFactory.fetchJedisConnector(JedisConnectionFactory.java:194) ~[spring-data-redis-1.8.18.RELEASE.jar:na] ... 25 common frames omitted Caused by: redis.clients.jedis.exceptions.JedisConnectionException: Failed connecting to host localhost:6379 at redis.clients.jedis.Connection.connect(Connection.java:207) ~[jedis-2.9.1.jar:na] at redis.clients.jedis.BinaryClient.connect(BinaryClient.java:101) ~[jedis-2.9.1.jar:na] at redis.clients.jedis.BinaryJedis.connect(BinaryJedis.java:1844) ~[jedis-2.9.1.jar:na] at redis.clients.jedis.JedisFactory.makeObject(JedisFactory.java:106) ~[jedis-2.9.1.jar:na] at org.apache.commons.pool2.impl.GenericObjectPool.create(GenericObjectPool.java:888) ~[commons-pool2-2.4.3.jar:2.4.3] at org.apache.commons.pool2.impl.GenericObjectPool.borrowObject(GenericObjectPool.java:432) ~[commons-pool2-2.4.3.jar:2.4.3] at org.apache.commons.pool2.impl.GenericObjectPool.borrowObject(GenericObjectPool.java:361) ~[commons-pool2-2.4.3.jar:2.4.3] at redis.clients.util.Pool.getResource(Pool.java:49) ~[jedis-2.9.1.jar:na] ... 28 common frames omitted Caused by: java.net.ConnectException: Connection refused: connect at java.net.DualStackPlainSocketImpl.waitForConnect(Native Method) ~[na:1.8.0\_144] at java.net.DualStackPlainSocketImpl.socketConnect(DualStackPlainSocketImpl.java:85) ~[na:1.8.0\_144] at java.net.AbstractPlainSocketImpl.doConnect(AbstractPlainSocketImpl.java:350) ~[na:1.8.0\_144] at java.net.AbstractPlainSocketImpl.connectToAddress(AbstractPlainSocketImpl.java:206) ~[na:1.8.0\_144] at java.net.AbstractPlainSocketImpl.connect(AbstractPlainSocketImpl.java:188) ~[na:1.8.0\_144] at java.net.PlainSocketImpl.connect(PlainSocketImpl.java:172) ~[na:1.8.0\_144] at java.net.SocksSocketImpl.connect(SocksSocketImpl.java:392) ~[na:1.8.0\_144] at java.net.Socket.connect(Socket.java:589) ~[na:1.8.0\_144] at redis.clients.jedis.Connection.connect(Connection.java:184) ~[jedis-2.9.1.jar:na] ... 35 common frames omitted

**场景描述**

springboot项目用redis解决session共享问题，启动项目报错。

springboot版本：

<parent> <groupId>org.springframework.boot</groupId> <artifactId>spring-boot-starter-parent</artifactId> <version>1.5.19.RELEASE</version> <relativePath/> <!-- lookup parent from repository --> </parent>

redis版本

[15332] 12 Mar 15:59:54.307 # Server started, Redis version 2.6.12 [15332] 12 Mar 15:59:54.308 \* The server is now ready to accept connections on port 6379

**异常原因**

redis服务器版本和jar包版本不一致造成的

maven仓库的 spring-boot-starter-redis1.3.8的只支持2.7.3 的redis

**解决方案**

更换redis服务器版本为springboot兼容的版本。这里更换为3.0。

[25072] 12 Mar 16:01:52.065 # Server started, Redis version 3.0.501 [25072] 12 Mar 16:01:52.065 \* The server is now ready to accept connections on port 6379

## 待整理

redis与java相连

设置redis中的配置文件redis.conf中的bind 为0.0.0.0

添加maven依赖

<!--redis-->

<dependency>

<groupId>redis.clients</groupId>

<artifactId>jedis</artifactId>

<version>2.9.0</version>

</dependency>

创建对象 Jedis jedis = new Jedis("192.168.181.129",6379);

使用该对象调用redis中的方法即可，在java中的方法名与redis中源码的方法名一样

redis与spring相连

1.添加maven依赖 spring-context

2.添加配置文件 spring-redis.xml

2.1连接池的配置 <bean id="jedicPoolConfig" class="redis.clients.jedis.JedisPoolConfig">

<!--无资源时等待时长-->

<property name="maxWaitMillis" value="5000"/>

<!--最大空闲连接数-->

<property name="maxIdle" value="10"/>

<!--最小空闲连接数-->

<property name="minIdle" value="5"/>

<!--最大连接数-->

<property name="maxTotal" value="20"/>

</bean>

2.2配置连接池 构造方法注入连接池

<bean id="jedisPool" class="redis.clients.jedis.JedisPool" destroy-method="close">

<constructor-arg name="host" value="127.0.0.1"/>

<constructor-arg name="port" value="6379"/>

<constructor-arg name="poolConfig" ref="jedicPoolConfig"/>

</bean>

3.书写类，以测试类为例

1.添加spring-test依赖交给spring管理

2.在测试类的头上使用注解的形式

1.将测试类交给spring @RunWith(SpringJUnit4ClassRunner.class)

2.读取配置文件 @ContextConfiguration(locations = "classpath:spring-redis.xml")

3.注入连接池@Autowired private JedisPool jedisPool;

4.获取Jedis对象 Jedis jedis = jedisPool.getResource();

方法名依然和redis源码中的方法名一样

存储对象：将对象序列化-->变成字符串(json)

1.导入依赖 gson

存对象：2.String jsonuser = new Gson().toJson(user);

3.jedis.set("user:1001"，json)

取对象：4String json = jedis.get("user:1001");

User user == new Gson().fromJson(json,User.class);

spring-data与redis连接 spring-data其实是一个综合项目 是一个spring的项目

添加依赖 spring-data-redis 1.8.0.RELEASE 要使它所依赖的spring的jar的版本低于spring的版本(只能低。不能高)

spring-tx spring-oxm spring-context-support 4.3.14.RELEASE

在spring-data-redis依赖中要排除spring-context-support 和spring-oxm

写配置文件

配置jedisConnectionFactory <bean id="jedisConnectionFactory" class="org.springframework.data.redis.connection.jedis.JedisConnectionFactory">

<property name="hostName" value="192.168.248.131"/>

<property name="port" value="6379"/>

<property name="usePool" value="true"/>

<property name="poolConfig" ref="jedicPoolConfig"/>

</bean>

连接池的配置 <bean id="jedicPoolConfig" class="redis.clients.jedis.JedisPoolConfig">

<!--无资源时等待时长-->

<property name="maxWaitMillis" value="5000"/>

<!--最大空闲连接数-->

<property name="maxIdle" value="10"/>

<!--最小空闲连接数-->

<property name="minIdle" value="5"/>

<!--最大连接数-->

<property name="maxTotal" value="20"/>

</bean>

配置模板 <bean id="redisTemplate" class="org.springframework.data.redis.core.RedisTemplate">

<property name="connectionFactory" ref="jedisConnectionFactory"/>

<property name="enableTransactionSupport" value="true"/>

</bean>

以测试类为例:

1.注入RedisTemplate @Autowired private RedisTemplate redisTemplate;

在注入的时候，可以使用setXXX方法注入，这样可以写一些额外的方法，比如键的序列化和值的序列化

private RedisTemplate redisTemplate;

@Autowired

public void setRedisTemplate(RedisTemplate redisTemplate){

this.redisTemplate = redisTemplate;

//写明键和值的规则，使用序列化成字符串，方便在redis源码读取时方便读取

this.redisTemplate.setKeySerializer(new StringRedisSerializer());

this.redisTemplate.setValueSerializer(new StringRedisSerializer());

}

2.直接使用即可，中间需要先实现接口

例：redisTemplate.opsForValue().set("hello","hello data");

在存储对象的时候，1.依然还是先转化为json对象，然后再转化为字符串

2.在set方法里指定值的存储方式 this.redisTemplate.setValueSerializer(new Jackson2JsonRedisSerializer<User>(User.class));

添加依赖 jackson-databind

3.不需要关闭，自己会自动关闭

使用Protostufflo进行高效序列化

1.添加依赖 <dependency>

<groupId>io.protostuff</groupId>

<artifactId>protostuff-core</artifactId>

<version>1.5.3</version>

</dependency>

<dependency>

<groupId>io.protostuff</groupId>

<artifactId>protostuff-runtime</artifactId>

<version>1.5.3</version>

</dependency>

2. 其他过程不变

User user = new User(1004,"康熙","北京",45);

//序列化

Schema<User> userSchema = RuntimeSchema.getSchema(User.class);

byte[] bytes = ProtobufIOUtil.toByteArray(user,userSchema,

LinkedBuffer.allocate(LinkedBuffer.DEFAULT\_BUFFER\_SIZE));

jedis.set("user:1004".getBytes(),bytes);

//反序列化

byte[] bytes1 = jedis.get("user:1004".getBytes());

Schema<User> userSchema1 = RuntimeSchema.getSchema(User.class);

User user1 = new User();

ProtobufIOUtil.mergeFrom(bytes,user1,userSchema1);

java连接redis集群

JedisCluster jedisCluster = null;

@Before

public void init(){

//连接池的配置

GenericObjectPoolConfig config = new GenericObjectPoolConfig();

config.setMaxTotal(10);

config.setMinIdle(5);

//集群主机集合

Set<HostAndPort> hostAndPorts = new HashSet<HostAndPort>();

hostAndPorts.add(new HostAndPort("192.168.248.133",6001));

hostAndPorts.add(new HostAndPort("192.168.248.133",6002));

hostAndPorts.add(new HostAndPort("192.168.248.134",6001));

hostAndPorts.add(new HostAndPort("192.168.248.134",6002));

hostAndPorts.add(new HostAndPort("192.168.248.135",6001));

hostAndPorts.add(new HostAndPort("192.168.248.135",6002));

jedisCluster = new JedisCluster(hostAndPorts,config);

}

@Test

public void set(){

jedisCluster.set("test","hello cluster");

String test = jedisCluster.get("test");

System.out.println(test);

}

@After

public void destory() throws IOException {

jedisCluster.close();

}

遇到出错，建议重新重新配一下集群 分配键值

spring jedis cluster

1.增加配置文件

1.1配置主机和端口对象<bean id="cluster133" class="redis.clients.jedis.HostAndPort">

配置6个主机(最少) <constructor-arg name="host" value="192.168.248.133"/>

<constructor-arg name="port" value="6001"/>

</bean>

1.2 创建jedisCluster对象 需要连接池的配置(在xml中)

<bean id="jedisCluster" class="redis.clients.jedis.JedisCluster">

<constructor-arg name="nodes">

<set>

<ref bean="cluster1351"/>

<ref bean="cluster1352"/>

<ref bean="cluster1342"/>

<ref bean="cluster1341"/>

<ref bean="cluster1332"/>

<ref bean="cluster1331"/>

</set>

</constructor-arg>

<constructor-arg name="poolConfig" ref="jedicPoolConfig"/>

</bean>

2.以测试类为例

注入JedisCluster对象，直接使用方法即可

spring-data 连接jedis cluster

<!--配置Cluster-->

<bean id="clusterConfiguration" class="org.springframework.data.redis.connection.RedisClusterConfiguration">

<property name="clusterNodes">

<list>

<bean class="org.springframework.data.redis.connection.RedisNode">

<constructor-arg name="host" value="192.168.248.133"/>

<constructor-arg name="port" value="6001"/>

</bean>

<bean class="org.springframework.data.redis.connection.RedisNode">

<constructor-arg name="host" value="192.168.248.133"/>

<constructor-arg name="port" value="6002"/>

</bean>

<bean class="org.springframework.data.redis.connection.RedisNode">

<constructor-arg name="host" value="192.168.248.134"/>

<constructor-arg name="port" value="6001"/>

</bean>

<bean class="org.springframework.data.redis.connection.RedisNode">

<constructor-arg name="host" value="192.168.248.134"/>

<constructor-arg name="port" value="6002"/>

</bean>

<bean class="org.springframework.data.redis.connection.RedisNode">

<constructor-arg name="host" value="192.168.248.135"/>

<constructor-arg name="port" value="6001"/>

</bean>

<bean class="org.springframework.data.redis.connection.RedisNode">

<constructor-arg name="host" value="192.168.248.135"/>

<constructor-arg name="port" value="6002"/>

</bean>

</list>

</property>

</bean>

<!--连接池的配置-->

<bean id="jedicPoolConfig" class="redis.clients.jedis.JedisPoolConfig">

<!--无资源时等待时长-->

<property name="maxWaitMillis" value="5000"/>

<!--最大空闲连接数-->

<property name="maxIdle" value="10"/>

<!--最小空闲连接数-->

<property name="minIdle" value="5"/>

<!--最大连接数-->

<property name="maxTotal" value="20"/>

</bean>

<!--配置JedisConnectionFactory-->

<bean id="jedisConnectionFactory" class="org.springframework.data.redis.connection.jedis.JedisConnectionFactory">

<!--配置集群节点-->

<constructor-arg name="clusterConfig" ref="clusterConfiguration"/>

<property name="usePool" value="true"/>

<property name="poolConfig" ref="jedicPoolConfig"/>

</bean>

<!--配置RedisTemplate模板-->

<bean id="redisTemplate" class="org.springframework.data.redis.core.RedisTemplate">

<property name="connectionFactory" ref="jedisConnectionFactory"/>

<property name="enableTransactionSupport" value="true"/>

</bean>

使用方法不变

## 总结

|  |
| --- |
| nosql  什么是nosql  nosql的作用  nosql的分类  redis  redis的由来  什么是redis  redis的应用场景  安装redis  简单应用  java操作redis(jedis)  redis中类型  redis中通用命令  redis的特性  nosql  no only sql 不仅仅是sql  一般称之为非关系型数据库  web2.0存在一些问题:(3高)  1.高并发  2.大数据  3.高扩展,高可用  分类:  key/value的格式  文档型格式  列类型  图  redis是使用c语言开发的一个高性能键值对的数据库  支持的数据类型如下  String(★)  hash(理解)  list  set  sortedSet(zset)  redis的安装  1.下载redis  2.上传到linux  3.安装redis  mkdir /usr/local/redis  mv /root/redis.tar /usr/local/redis  cd /usr/local/redis  tar -xvf redis.tar  4.编译redis 依赖 gcc  yum install gcc-c++  make  5.安装redis  make PREFIX=/usr/local/redis install  6.配置  复制一个redis.conf 到bin目录下  启动服务器的方式1:  前台启动的方式:  cd /usr/local/redis/bin  ./redis-server redis.conf  后台的方式:  配置一下redis.conf  修改:daemonize yes  保存退出  启动客户端  简单的方式:  ./redis-cli #连接本地端口号为 6479的服务器  推荐的方式  ./redis-cli -h 连接ip -p 端口号    redis的停止  方式1:通过kill -9 进程号(不推荐)  方式2:通过客户端发送命令  ./redis-cli -h ip -p port shutdown  数据类型  string  掌握的操作:  赋值  格式: set key value  例如:set username tom  取值  格式: get key  例如: get username    先获取再设置  getset key value  例如: getset username jack    删  del key  例如: del d  了解:  对于数字类型 自增和自减  incr key ++  decr key --  增加或减少指定的数量  incrby key int  decrby key int  拼接字符串  append key value    list  赋值:  左边:lpush key value value2 value3  右边:rpush key value value2 value3  取值:  左边:lpop key  右边:rpush key  获取所有元素  lrange 0 -1  获取元素的个数  llen key  扩展:  lpushx key value :若有则添加 若没有则不添加  rpushx key value :若有则添加 若没有则不添加  lrem key count value:从左边移除count个value  若count>0 :从左边移除count个value  若count<0 :从右边移除count个value  若count=0 :从右边移除所有的value  lset key index value  设置链表中指定索引的元素值 0 代表是第一个  -1代表的是最后一个  hash:  了解  又存入一个map集合  user username tom  age 18  sex 1  存值:  存入一个值  hset key subkey subvalue  存入多个值  hmset key subkey1 subvalue1 subkey2 subvalue2    获取:  获取一个值  hget key subkey  获取多个值  hmget key subkey1 subkey2    移除值:  hdel key subkey subkey  给一个key添加指定的数字  hincrby key subkey int    set  添加  sadd key value1 valuse2  删除  srem key value1 valuse2  获取  smembers key  判断是否是是set中的一员  sismember key value  运算  差集: sdiff s1 s2  交集: sinter s1 s2  并集: sunion s3 s4  获取数量  scard key  srandmember key:随机获取一个    sortedSet  添加元素  zadd key score m1 score m2  获取元素  zscore key m:获取指定成员的得分  zcard key:获取key的长度  删除元素  zrem  通用的操作:  keys \* :查看所有的key  del key:删除指定的key  exists key:判断一个key是否存在  rename oldkey newkey:重命名  expire key 秒数:  ttl key :查看一个key剩余存活时间  -1:持久存活  -2:不存在  type 判断一个可以属于什么类型 |

# Ehcache

<https://www.cnblogs.com/chenmc/p/9273714.html>

|  |
| --- |
| @Override  @Cacheable(value = "MS\_Cache", key = "'user:'+#id")  public MsProductInfo queryProductById(int id) {  System.out.println("看到这条消息证明去查询数据库");  return msProductInfoService.queryProductById(id);  }  @Override  @CachePut(value = "MS\_Cache", key = "'user:'+#id")  public MsProductInfo upadteProductById( MsProductInfo MsProductInfo) {  msProductInfoService.updateMsProductInfo(MsProductInfo);  return msProductInfoService.queryProductById(MsProductInfo.getId());  } |

# 123

在[**Java**](http://lib.csdn.net/base/javase)项目广泛的使用。它是一个开源的、设计于提高在数据从RDBMS中取出来的高花费、高延迟采取的一种缓存方案。正因为Ehcache具有健壮性（基于java开发）、被认证（具有apache 2.0  license）、充满特色（稍后会详细介绍），所以被用于大型复杂分布式web application的各个节点中。

什么特色？

1.  够快

Ehcache的发行有一段时长了，经过几年的努力和不计其数的性能[**测试**](http://lib.csdn.net/base/softwaretest)，Ehcache终被设计于large, high concurrency systems.

2. 够简单

开发者提供的接口非常简单明了，从Ehcache的搭建到运用运行仅仅需要的是你宝贵的几分钟。其实很多开发者都不知道自己用在用Ehcache，Ehcache被广泛的运用于其他的开源项目

比如：[**hibernate**](http://lib.csdn.net/base/javaee)

3.够袖珍

关于这点的特性，官方给了一个很可爱的名字small foot print ，一般Ehcache的发布版本不会到2M，V 2.2.3  才 668KB。

4. 够轻量

核心程序仅仅依赖slf4j这一个包，没有之一！

5.好扩展

Ehcache提供了对[**大数据**](http://lib.csdn.net/base/hadoop)的内存和硬盘的存储，最近版本允许多实例、保存对象高灵活性、提供LRU、LFU、FIFO淘汰[**算法**](http://lib.csdn.net/base/datastructure)，基础属性支持热配置、支持的插件多

6.监听器

缓存管理器监听器 （CacheManagerListener）和 缓存监听器（CacheEvenListener）,做一些统计或数据一致性广播挺好用的

如何使用？

够简单就是Ehcache的一大特色，自然用起来just so easy!

贴一段基本使用代码

CacheManager manager = CacheManager.newInstance("src/config/ehcache.xml");

Ehcache cache = new Cache("testCache", 5000, false, false, 5, 2);

cacheManager.addCache(cache);

       name:缓存名称。

       maxElementsInMemory：缓存最大个数。

       eternal:对象是否永久有效，一但设置了，timeout将不起作用。

       timeToIdleSeconds：设置对象在失效前的允许闲置时间（单位：秒）。仅当eternal=false对象不是永久有效时使用，可选属性，默认值是0，也就是可闲置时间无穷大。

       timeToLiveSeconds：设置对象在失效前允许存活时间,最大时间介于创建时间和失效时间之间。仅当eternal=false对象不是永久有效时使用，默认是0.，也就是对象存活时 间无穷大。

overflowToDisk：当内存中对象数量达到maxElementsInMemory时，Ehcache将会对象写到磁盘中。

diskSpoolBufferSizeMB：这个参数设置DiskStore（磁盘缓存）的缓存区大小。默认是30MB。每个Cache都应该有自己的一个缓冲区。

maxElementsOnDisk：硬盘最大缓存个数。

diskPersistent： 是否缓存虚拟机重启期数 据 Whether the disk store persists between restarts of the Virtual Machine. The default value is false.

diskExpiryThreadIntervalSeconds：磁盘失效线程运行时间间隔，默认是120秒。

memoryStoreEvictionPolicy：当达到maxElementsInMemory限制时，Ehcache将会根据指定的策略去清理内存。默认策略是LRU。你可以设置为 FIFO或是LFU。

       clearOnFlush：内存数量最大时是否清除。

**memcache**

memcache 是一种高性能、分布式对象缓存系统，最初设计于缓解动态网站[**数据库**](http://lib.csdn.net/base/mysql)加载数据的延迟性，你可以把它想象成一个大的内存HashTable，就是一个key-value键值缓存。Danga Interactive为了LiveJournal所发展的，以BSD license释放的一套开放源代码软件。

1.依赖

memcache [**C语言**](http://lib.csdn.net/base/c)所编写，依赖于最近版本的GCC和libevent。GCC是它的编译器，同事基于libevent做socket io。在安装memcache时保证你的系统同事具备有这两个环境。

2.多线程支持

memcache支持多个cpu同时工作，在memcache安装文件下有个叫threads.txt中特别说明，By default, memcached is compiled as a single-threaded application.默认是单线程编译安装，如果你需要多线程则需要修改./configure --enable-threads，为了支持多核系统，前提是你的系统必须具有多线程工作模式。开启多线程工作的线程数默认是4，如果线程数超过cpu数 容易发生操作死锁的概率。结合自己业务模式选择才能做到物尽其用。

3.高性能

通过libevent完成socket 的通讯，理论上性能的瓶颈落在网卡上。

简单安装：

1.分别把memcached和libevent下载回来，放到 /tmp 目录下：

# cd /tmp

# wget http://www.danga.com/memcached/dist/memcached-1.2.0.tar.gz

# wget http://www.monkey.org/~provos/libevent-1.2.tar.gz

2.先安装libevent：

# tar zxvf libevent-1.2.tar.gz

# cd libevent-1.2

# ./configure -prefix=/usr

# make （如果遇到提示gcc 没有安装则先安装gcc)

# make install

3.测试libevent是否安装成功：

# ls -al /usr/lib | grep libevent

lrwxrwxrwx 1 root root 21 11?? 12 17:38 libevent-1.2.so.1 -> libevent-1.2.so.1.0.3

-rwxr-xr-x 1 root root 263546 11?? 12 17:38 libevent-1.2.so.1.0.3

-rw-r-r- 1 root root 454156 11?? 12 17:38 libevent.a

-rwxr-xr-x 1 root root 811 11?? 12 17:38 libevent.la

lrwxrwxrwx 1 root root 21 11?? 12 17:38 libevent.so -> libevent-1.2.so.1.0.3

还不错，都安装上了。

4.安装memcached，同时需要安装中指定libevent的安装位置：

# cd /tmp

# tar zxvf memcached-1.2.0.tar.gz

# cd memcached-1.2.0

# ./configure -with-libevent=/usr

# make

# make install

如果中间出现报错，请仔细检查错误信息，按照错误信息来配置或者增加相应的库或者路径。

安装完成后会把memcached放到 /usr/local/bin/memcached ，

5.测试是否成功安装memcached：

# ls -al /usr/local/bin/mem\*

-rwxr-xr-x 1 root root 137986 11?? 12 17:39 /usr/local/bin/memcached

-rwxr-xr-x 1 root root 140179 11?? 12 17:39 /usr/local/bin/memcached-debug

启动memcache服务

启动Memcached服务：

1.启动Memcache的服务器端：

# /usr/local/bin/memcached -d -m 8096 -u root -l 192.168.77.105 -p 12000 -c 256 -P /tmp/memcached.pid

-d选项是启动一个守护进程，

-m是分配给Memcache使用的内存数量，单位是MB，我这里是8096MB，

-u是运行Memcache的用户，我这里是root，

-l是监听的服务器IP地址，如果有多个地址的话，我这里指定了服务器的IP地址192.168.77.105，

-p是设置Memcache监听的端口，我这里设置了12000，最好是1024以上的端口，

-c选项是最大运行的并发连接数，默认是1024，我这里设置了256，按照你服务器的负载量来设定，

-P是设置保存Memcache的pid文件，我这里是保存在 /tmp/memcached.pid，

2.如果要结束Memcache进程，执行：

# cat /tmp/memcached.pid 或者 ps -aux | grep memcache   （找到对应的进程id号）

# kill 进程id号

也可以启动多个守护进程，不过端口不能重复。

 memcache 的连接

telnet  ip   port

注意连接之前需要再memcache服务端把memcache的防火墙规则加上

-A RH-Firewall-1-INPUT -m state --state NEW -m tcp -p tcp --dport 3306 -j ACCEPT

重新加载防火墙规则

service iptables restart

OK ,现在应该就可以连上memcache了

在客户端输入stats 查看memcache的状态信息

pid              memcache服务器的进程ID

uptime      服务器已经运行的秒数

time           服务器当前的unix时间戳

version     memcache版本

pointer\_size         当前[**操作系统**](http://lib.csdn.net/base/operatingsystem)的指针大小（32位系统一般是32bit）

rusage\_user          进程的累计用户时间

rusage\_system    进程的累计系统时间

curr\_items            服务器当前存储的items数量

total\_items           从服务器启动以后存储的items总数量

bytes                       当前服务器存储items占用的字节数

curr\_connections        当前打开着的连接数

total\_connections        从服务器启动以后曾经打开过的连接数

connection\_structures          服务器分配的连接构造数

cmd\_get get命令          （获取）总请求次数

cmd\_set set命令          （保存）总请求次数

get\_hits          总命中次数

get\_misses        总未命中次数

evictions     为获取空闲内存而删除的items数（分配给memcache的空间用满后需要删除旧的items来得到空间分配给新的items）

bytes\_read    读取字节数（请求字节数）

bytes\_written     总发送字节数（结果字节数）

limit\_maxbytes     分配给memcache的内存大小（字节）

threads         当前线程数

### redis

[**Redis**](http://lib.csdn.net/base/redis)是 在memcache之后编写的，大家经常把这两者做比较，如果说它是个key-value store 的话但是它具有丰富的数据类型，我想暂时把它叫做缓存数据流中心，就像现在物流中心那样，order、package、store、 classification、distribute、end。现在还很流行的LAMP [**PHP**](http://lib.csdn.net/base/php)[**架构**](http://lib.csdn.net/base/architecture) 不知道和 redis+[**MySQL**](http://lib.csdn.net/base/mysql) 或者 redis + [**MongoDB**](http://lib.csdn.net/base/mongodb)的性能比较（听群里的人说mongodb分片不稳定）。

先说说reidis的特性

1. 支持持久化

     redis的本地持久化支持两种方式：RDB和AOF。RDB 在redis.conf配置文件里配置持久化触发器，AOF指的是redis没增加一条记录都会保存到持久化文件中（保存的是这条记录的生成命令），如果 不是用redis做DB用的话还会不要开AOF ，数据太庞大了，重启恢复的时候是一个巨大的工程！

2.丰富的数据类型

    redis 支持 String 、Lists、sets、sorted sets、hashes 多种数据类型,新浪微博会使用redis做nosql主要也是它具有这些类型，时间排序、职能排序、我的微博、发给我的这些功能List 和 sorted set

   的强大操作功能息息相关

3.高性能

   这点跟memcache很想象，内存操作的级别是毫秒级的比硬盘操作秒级操作自然高效不少，较少了磁头寻道、数据读取、页面交换这些高开销的操作！这也是NOSQL冒出来的原因吧，应该是高性能

  是基于RDBMS的衍生产品，虽然RDBMS也具有缓存结构，但是始终在app层面不是我们想要的那么操控的。

4.replication

    redis提供主从复制方案，跟mysql一样增量复制而且复制的实现都很相似，这个复制跟AOF有点类似复制的是新增记录命令，主库新增记录将新增脚本 发送给从库，从库根据脚本生成记录，这个过程非常快，就看网络了，一般主从都是在同一个局域网，所以可以说redis的主从近似及时同步，同事它还支持一 主多从，动态添加从库，从库数量没有限制。 主从库搭建，我觉得还是采用网状模式，如果使用链式（master-slave-slave-slave-slave·····）如果第一个slave出 现宕机重启，首先从master 接收 数据恢复脚本，这个是阻塞的，如果主库数据几TB的情况恢复过程得花上一段时间，在这个过程中其他的slave就无法和主库同步了。

5.更新快

   这点好像从我接触到redis到目前为止 已经发了大版本就4个，小版本没算过。redis作者是个非常积极的人，无论是邮件提问还是论坛发帖，他都能及时耐心的为你解答，维护度很高。有人维护的 话，让我们用的也省心和放心。目前作者对redis 的主导开发方向是redis的集群方向。

redis的安装

redis的安装其实还是挺简单的，总的来说就三步：下载tar包，解压tar包，安装。

不过最近我在2.6.7后用centos 5.5 32bit 时碰到一个安装问题，下面我就用图片分享下安装过程碰到的问题，在redis 文件夹内执行make时有个如下的错 undefined reference to '\_\_sync\_add\_and\_fetch\_4'

上网找了了好多最后在  https://github.com/antirez/redis/issues/736 找到解决方案，write CFLAGS= -march=i686 on src/Makefile head!

记得要把刚安装失败的文件删除，重新解压新的安装文件，修改Makefile文件，再make安装。就不会发现原来那个错误了

关于redis的一些属性注释和基本类型操作在上一篇redis 的开胃菜有详细的说明，这里就不再重复累赘了（实质是想偷懒 ，哈哈！）

最后，把memcache和redis放在一起不得不会让人想到两者的比较，谁快谁好用啊，群里面已经为这个事打架很久了，我就把我看到的在这里跟大家分享下。

在别人发了一个memcache性能比redis好很多后，redis 作者 antirez 发表了一篇博文，主要是说到如何给redis 和 memcache 做压力测试，文中讲到有个人说许多开源软件都应该丢进厕所，因为他们的压力测试脚本太2了，作者对这个说明了一番。redis  vs  memcache is  definitely an apple to apple comparison。 呵呵，很明确吧，两者的比较是不是有点鸡蛋挑骨头的效果，作者在相同的运行环境做了三次测试取多好的值，得到的结果如下图:

需要申明的是此次测试在单核心处理的过程的数据，memcache是支持多核心多线程操作的（默认没开）所以在默认情况下上图具有参考意义，若然则 memcache快于redis。那为什么redis不支持多线程多核心处理呢？作者也发表了一下自己的看法，首先是多线程不变于bug的修复，其实是不 易软件的扩展，还有数据一致性问题因为redis所有的操作都是原子操作，作者用到一个词nightmare 噩梦，呵呵！  当然不支持多线程操作，肯定也有他的弊端的比如性能想必必然差，作者从2.2版本后专注redis cluster的方向开发来缓解其性能上的弊端，说白了就是纵向不行，横向提高。

**应用场景：**

**ehcache直接在jvm虚拟机中缓存，速度快，效率高；但是缓存共享麻烦，集群分布式应用不方便。**

**redis是通过socket访问到缓存服务，效率比ecache低，比数据库要快很多，处理集群和分布式缓存方便，有成熟的方案。**

**如果是单个应用或者对缓存访问要求很高的应用，用ehcache。**

**如果是大型系统，存在缓存共享、分布式部署、缓存内容很大的，建议用redis。**

**补充下：ehcache也有缓存共享方案，不过是通过RMI或者Jgroup多播方式进行广播缓存通知更新，缓存共享复杂，维护不方便；简单的共享可以，但是涉及到缓存恢复，大数据缓存，则不合适  
  
redis和memcached相比的独特之处：  
1、redis可以用来做存储(storage),而memcached是用来做缓存(cache)  
这个特点主要因为其有持久化功能  
  
2、redis中存储的数据有多种结构，而memcached存储的数据只有一种类型“字符串”**第二种理解:

第一：两者之间的介绍

Redis：属于独立的运行程序，需要单独安装后，使用JAVA中的Jedis来操纵。因为它是独立，所以如果你写个单元测试程序，放一些数据在Redis中，然后又写一个程序去拿数据，那么是可以拿到这个数据的。，

ehcache：与Redis明显不同，它与java程序是绑在一起的，java程序活着，它就活着。譬如，写一个独立程序放数据，再写一个独立程序拿数据，那么是拿不到数据的。只能在独立程序中才能拿到数据。

第二：使用及各种配置：

两者都可以集群：

1.Redis可以做主从来集群，例如，在A电脑上装个Redis，作为主库；在其他电脑上装Redis，作为从库；这样主库拥有读和写的功能，而从库只拥有读的功能。每次主库的数据都会同步到从库中。

1.默认方式启动

|  |  |
| --- | --- |
| 1  2  3  4  5 | Linux下使用Redis  安装：从官网上下载tar.gz格式的包，然后使用tar zxvf redis-2.8.24.tar.gz命令解压，然后进入Redis文件夹目录下的src目录，使用make编译一下    1.开启：进入/usr/local/redis-3.2.1/src  然后./redis-server |

2.如果我们想修改端口，设置密码：那么得修改配置文件的redis.conf

port 6379                                         //端口修改

**requirepass redis123                  //设置密码  redis123为密码**

**配置主从：主库的配置文件不用修改，从库的配置文件需要修改，因为从库需要绑定主库，以便可以获取主库的数据**

**slaveof 192.168.1.100 6379                              //主库的IP地址和端口号  
masterauth redis123                                      //主库设定的密码**

**3.要让配置文件的属性生效，那么启动的redis的时候，要将配置文件加上去**

进入/usr/local/redis-3.2.1/src

**然后 ./redis-server  redis.conf**

**那么将成功的启动redis，如果没有加入配置的话，按照普通方式启动的话，端口仍然还是6379.**

**4.客户端连接远程的Redis  
第一步：在远程端处设置密码：config set requirepass 123       //123为密码  
第二步：可以在客户端登录  redis-cli.exe -h 114.215.125.42 -p 6379   
第三步：认证：auth 123                                       //123为密码  
本地端设置密码后，要使用密码登录；如果Redis重启的话，密码需要重新设置**

**5.主从配置后，为保证主库写的能力，一般不在主库做持久化，而是在从库做持久化：**

**主库配置：**

**将save注释，不使用rdb**

**# save 900 1  
# save 300 10  
# save 60 10000**

**appendonly no        不使用aof**

**从库配置：**

**save 900 1  
save 300 10  
save 60 10000**

**appendonly yes**

**这样做的优缺点：**

**优点：保证了主库写的能力。**

**缺点：主库挂掉后，重启主库，然后进行第一次写的动作后，主库会先生成rdb文件，然后传输给从库，从而覆盖掉从库原先的rdb文件，造成数据丢失。但是第二次写的时候，主库会以快照方式直接传数据给从库，不会重新生成rdb文件。**

**解决方案：先复制从库中的数据到主库后，再启动主库。**

使用：

引入jedis包

<dependency>

<groupId>redis.clients</groupId>

<artifactId>jedis</artifactId>

<version>2.7.3</version>

</dependency>

简单的写个类玩玩吧

[复制代码](javascript:void(0);)

public class RedisMain {

public static void main(String [] str)

{

Jedis jedis = new Jedis("114.215.125.42",6379);

jedis.auth("123"); //密码认证

System.out.println("Connection to server sucessfully");

//查看服务是否运行

jedis.set("user","namess");

// System.out.println("Server is running: "+jedis.ping());

System.out.println(jedis.get("user").toString());

jedis.set("user","name");

System.out.println(jedis.get("user"));

}

}

[复制代码](javascript:void(0);)

**Redis完毕**

下面说说Ehcache:

Ehcache的使用：

1.首先引入包

[复制代码](javascript:void(0);)

<dependency>

<groupId>net.sf.ehcache</groupId>

<artifactId>ehcache-core</artifactId>

<version>2.6.6</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

<version>1.6.6</version>

</dependency>

[复制代码](javascript:void(0);)

2.创建一个ehcache.xml文件，里面配置cache的信息，这个配置是包含了集群的配置：与192.168.93.129：40001的 机器集群了：Ip为192.168.93.129机子的配置要将rmiUrls对应的数据改为这个配置文件的机子的IP地址，和对应的缓存名字

[复制代码](javascript:void(0);)

<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="ehcache.xsd">

<cacheManagerPeerProviderFactory

class="net.sf.ehcache.distribution.RMICacheManagerPeerProviderFactory"

properties="peerDiscovery=manual,rmiUrls=//192.168.93.129:40001/demoCache"/> <!--另一台机子的ip缓存信息-->

<cacheManagerPeerListenerFactory class="net.sf.ehcache.distribution.RMICacheManagerPeerListenerFactory"

properties="hostName=localhost,port=40001,socketTimeoutMillis=2000" /> <!--hostName代表本机子的ip-->

<diskStore path="java.io.tmpdir"/>

<defaultCache

maxElementsInMemory="10000"

maxElementsOnDisk="0"

eternal="true"

overflowToDisk="true"

diskPersistent="false"

timeToIdleSeconds="0"

timeToLiveSeconds="0"

diskSpoolBufferSizeMB="50"

diskExpiryThreadIntervalSeconds="120"

memoryStoreEvictionPolicy="LFU"

>

<cacheEventListenerFactory

class="net.sf.ehcache.distribution.RMICacheReplicatorFactory"/>

</defaultCache>

<cache name="demoCache"

maxElementsInMemory="100"

maxElementsOnDisk="0"

eternal="false"

overflowToDisk="false"

diskPersistent="false"

timeToIdleSeconds="119"

timeToLiveSeconds="119"

diskSpoolBufferSizeMB="50"

diskExpiryThreadIntervalSeconds="120"

memoryStoreEvictionPolicy="FIFO"

>

<cacheEventListenerFactory class="net.sf.ehcache.distribution.RMICacheReplicatorFactory"/> <!--监听这个cache-->

</cache>

</ehcache>

[复制代码](javascript:void(0);)

配置完后写代码：

放数据：

[复制代码](javascript:void(0);)

@RequestMapping("/testehcache.do")

public void testehcache(HttpServletResponse response) throws IOException

{

URL url = getClass().getResource("ehcache.xml");

CacheManager singletonmanager = CacheManager.create(url);

Cache cache = singletonmanager.getCache("demoCache");

//使用缓存

Element element = new Element("key1", "value1");

cache.put(element);

cache.put(new Element("key2", "value2"));

response.getWriter().println("我存放了数据");

}

[复制代码](javascript:void(0);)

拿数据：

[复制代码](javascript:void(0);)

@RequestMapping("/getcache.do")

public void getcache(HttpServletResponse response) throws IOException

{

CacheManager singletonmanager = CacheManager.create();

Cache cache = singletonmanager.getCache("demoCache");

String one=cache.get("key1").getObjectValue().toString();

String two=cache.get("key2").getObjectValue().toString();

response.getWriter().println(one+two);

}

[复制代码](javascript:void(0);)

配置集群后，A机器放数据，在B机器上能拿到数据，B机器放数据，A机器也可以拿到数据