# 一、使用Spring-data-redis

## 1.引入依赖

|  |
| --- |
| <**dependency**>  <**groupId**>junit</**groupId**>  <**artifactId**>junit</**artifactId**>  <**version**>4.12</**version**> </**dependency**>   <**dependency**>  <**groupId**>org.springframework.data</**groupId**>  <**artifactId**>spring-data-redis</**artifactId**>  <**version**>1.8.8.RELEASE</**version**> </**dependency**> <**dependency**>  <**groupId**>org.springframework</**groupId**>  <**artifactId**>spring-test</**artifactId**>  <**version**>4.3.8.RELEASE</**version**> </**dependency**>  <**dependency**>  <**groupId**>redis.clients</**groupId**>  <**artifactId**>jedis</**artifactId**>  <**version**>2.9.0</**version**> </**dependency**> |

## 2.编写配置文件

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd"**>    *<!-- 配置redis连接池对象 -->* <**bean id="poolConfig" class="redis.clients.jedis.JedisPoolConfig"**>  *<!-- 最大空闲数 -->* <**property name="maxIdle" value="50"** />  *<!-- 最大连接数 -->* <**property name="maxTotal" value="100"** />  *<!-- 最大等待时间 -->* <**property name="maxWaitMillis" value="20000"** />  </**bean**>   *<!-- 配置redis连接工厂 -->* <**bean id="connectionFactory"  class="org.springframework.data.redis.connection.jedis.JedisConnectionFactory"**>  *<!-- 连接池配置 -->* <**property name="poolConfig" ref="poolConfig"** />  *<!-- 连接主机 -->* <**property name="hostName" value="192.168.2.128"** />  *<!-- 端口 -->* <**property name="port" value="6379"** />  *<!-- 密码 -->* <**property name="password" value="java1904"** />  </**bean**>   *<!-- 配置redis模板对象 -->* <**bean class="org.springframework.data.redis.core.RedisTemplate"**>  *<!-- 配置连接工厂 -->* <**property name="connectionFactory" ref="connectionFactory"** />  </**bean**>    </**beans**> |

## 3.编写测试类，注入RedisTemplate对象

|  |
| --- |
| @RunWith(SpringJUnit4ClassRunner.**class**) @ContextConfiguration(locations = **"classpath:spring-context.xml"**) **public class** TestSpringDataRedis {   @Autowired  **private** RedisTemplate **redisTemplate**; |

## 4.通过redisTemplate对象来操作redis中的五大数据类型

通过redisTemplate对象的opsFor\*\*\*().set()或get()来操作

# 二、spring-data-redis中的序列化

## 1.关于序列化

使用序列化器能够方便的来存放javabean类型的对象

默认情况下，使用redisTemplate进行操作时，键值对都会被序列化。因此，在没有指明redisTemplate默认的序列化器的情况下，存放的键值对都会使用Jdk序列化器作为默认的序列化器，来序列化键值对。

所以我们发现使用redisTemplate存放数据时，在redis中的数据都是被序列化过的。

如果改用StringRedisSerializer 字符串序列化器，那么就不会想jdk的序列化器那样生成序列化后的字符。

jdk: \xac\xed\x00\x05t\x00\x06target

string: target

实际上，我们往往将键用string的序列化器，值用默认的jdk序列化器，因为键使用string序列化器节省空间，值使用jdk序列化器可以存储对象。

## 2.如何配置redisTemplate里的序列化器

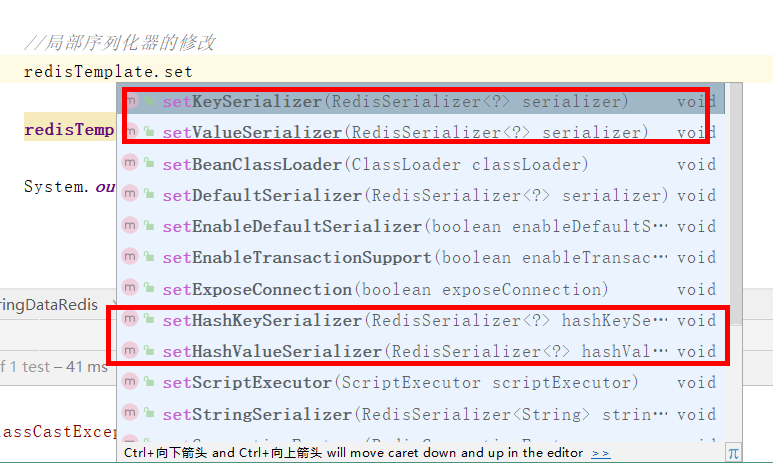
### 1）全局配置

在spring-context.xml中修改：

|  |
| --- |
| <**bean id="stringRedisSerializer" class="org.springframework.data.redis.serializer.StringRedisSerializer"**> </**bean**>  *<!-- 配置redis模板对象 -->* <**bean class="org.springframework.data.redis.core.RedisTemplate"**>  *<!-- 配置连接工厂 -->* <**property name="connectionFactory" ref="connectionFactory"** />  <**property name="keySerializer" ref="stringRedisSerializer"**/>  <**property name="valueSerializer" ref="stringRedisSerializer"**/>  <**property name="hashKeySerializer" ref="stringRedisSerializer"**/>  <**property name="hashValueSerializer" ref="stringRedisSerializer"**/> </**bean**> |

### 2）局部配置

使用redisTemplate的api来修改序列化器。



# 三、Redis中的事务

## 1. 在redis中操作事务：

如何开启事务： multi

如何提交事务： exec

## 2. 在spring-data-redis中操作事务

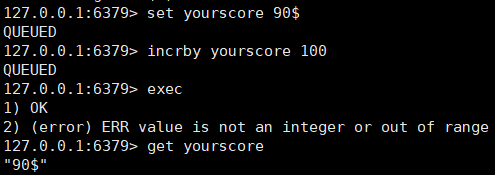
需用通过调用execute方法来操作事务

|  |
| --- |
| **redisTemplate**.execute(**new** SessionCallback() {  **public** Object execute(RedisOperations operations) **throws** DataAccessException {  operations.multi();*//开启事务* operations.opsForValue().set(**"money"**,**"20000"**);  operations.opsForValue().increment(**"money"**,10000);   operations.exec();*//提交事务* **return null**;  } }); |

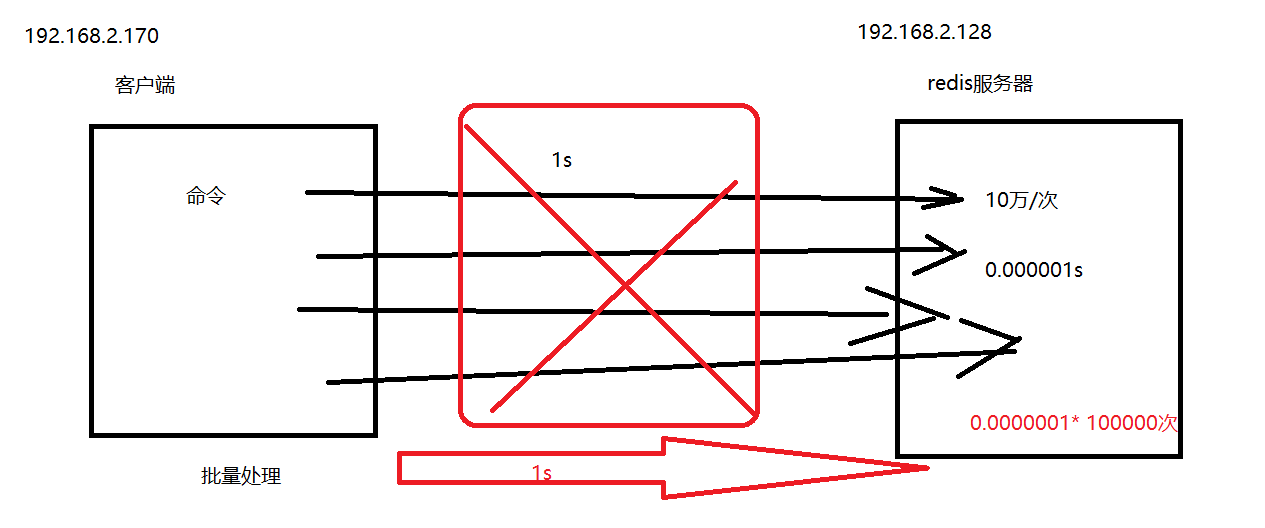
注意：

redis的事务并不严谨，如果出现命令格式正确，但数据类型错误的情况就会导致事务不回滚。





# 四、Redis中的流水线==批处理



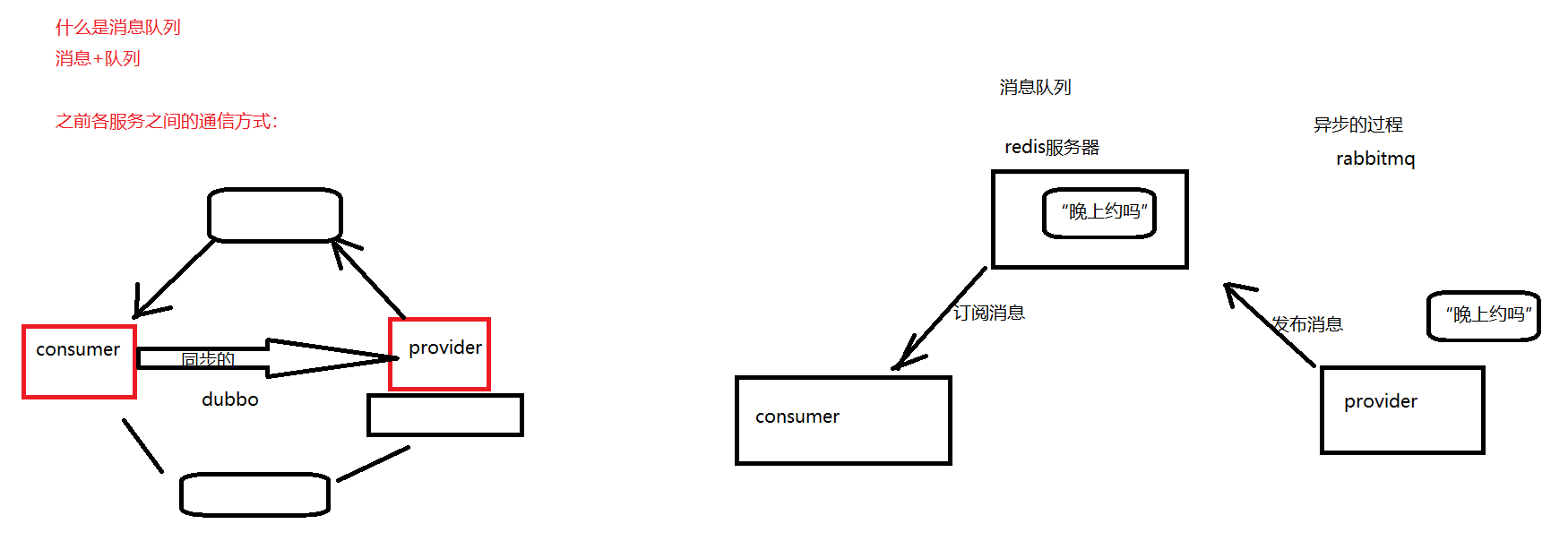
Redis中的流水线能够避免网络传输带来的性能瓶颈。让多条指令打包，一起发给redis，一次传输就够了，不会因多次传输造成的网络性能的瓶颈。

如何实现？**redisTemplate**.executePipelined()

|  |
| --- |
| @Test  **public void** testBatchPipelined(){   **long** start = System.*currentTimeMillis*();  *//* **redisTemplate**.executePipelined(**new** SessionCallback() {  **public** Object execute(RedisOperations operations) **throws** DataAccessException {   **for** (**int** i = 0; i < 20000; i++) {  operations.opsForValue().set(**"kk"**+i,**"vv"**+i);  } **return null**;  }  });**long** end = System.*currentTimeMillis*();  System.***out***.println(end-start);    } |

# 五、Redis中如何实现消息队列

## 1.消息队列的快速认识



## 2.具体的实现

### 1）在redis客户端中实现

消息的发布者：

publish 通道 消息

消息的订阅者：

subscribe 通道

### 2）在spring-data-redis中实现

步骤：

（1）创建一个监听类 实现MessageListener接口

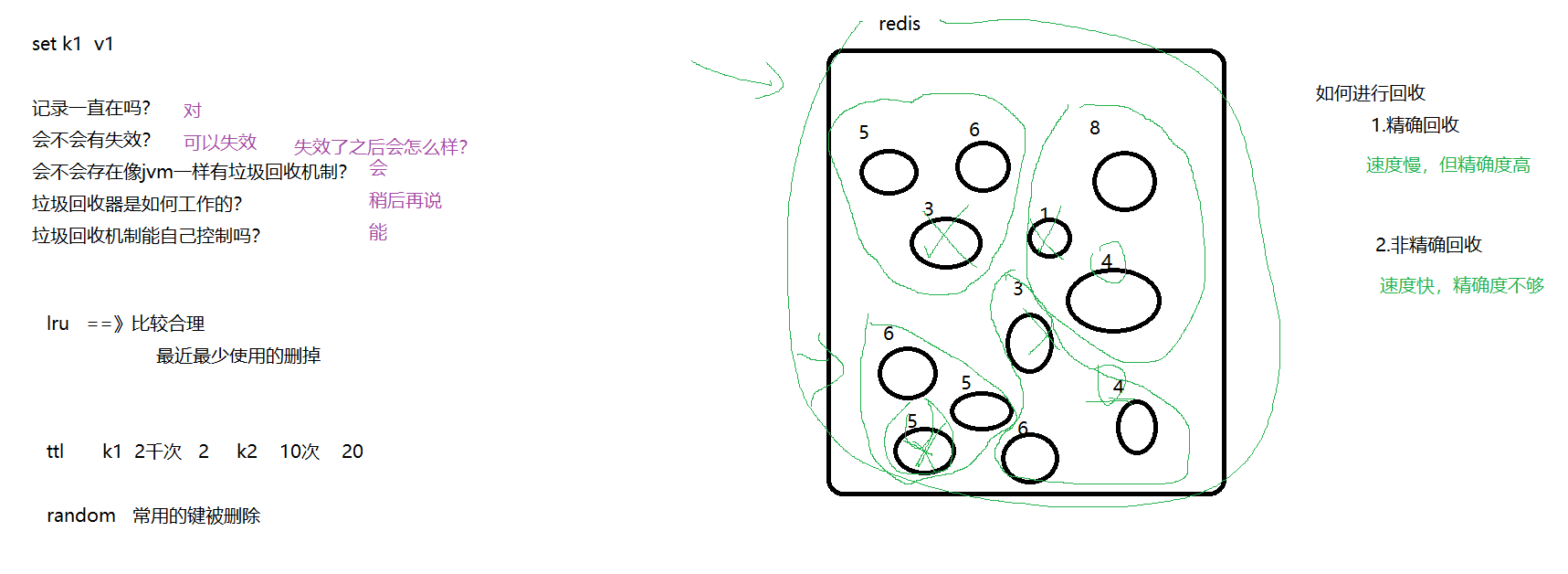
（2）重写方法，其中参数message是可以用来获取通道信息及消息信息

|  |
| --- |
| **public class** MyRedisListener **implements** MessageListener {  **public void** onMessage(Message message, **byte**[] pattern) {  **byte**[] channel = message.getChannel();  **try** {  String myChannel = **new** String(channel,**"utf-8"**);//获取通道的名字  String msg = **new** String(message.getBody(),**"utf-8"**);//获取通道的信息  System.***out***.println(**"通道："**+myChannel);  System.***out***.println(**"消息："**+msg);  } **catch** (UnsupportedEncodingException e) {  e.printStackTrace();  }  } } |

（3）在spring-context.xml中配置该监听类。

|  |
| --- |
| *<!--自己实现的监听器-->* <**bean id="myRedisListener" class="com.qf.listener.MyRedisListener"**></**bean**>  *<!-- 配置消息监听容器 -->* <**bean id="messageListenerContainer" class="org.springframework.data.redis.listener.RedisMessageListenerContainer"  destroy-method="destroy"**>  *<!-- redis连接工厂 -->* <**property name="connectionFactory" ref="connectionFactory"**/>  *<!-- 配置监听配置 -->* <**property name="messageListeners"**>  <**map**>  *<!-- 配置监听器 -->* <**entry key-ref="myRedisListener"**>  *<!-- 配置监听的渠道 -->* <**bean class="org.springframework.data.redis.listener.ChannelTopic"**>  *<!-- 配置要监听的渠道名字 -->* <**constructor-arg name="name" value="nba"**/>  </**bean**>  </**entry**>  </**map**>  </**property**> </**bean**> |

# 六、Redis的内存管理



## 1.设置redis中记录的超时时间

默认情况下，redis中的记录是永久有效的，可以通过设置其超时时间，等时间一到就失效。

做法：

### 1）在redis客户端中

ttl 键==》 查看当前记录的超时时间：

-1 永久有效

非-1 和-2的值： 还剩多少时间超时

-2 已超时

expire 键 秒值

设置当前键的超时时间

persist 键

让 当前记录永久有效。

### 2）在spring-data-redis中

|  |
| --- |
| **redisTemplate**.opsForValue().set(**"k1"**,**"v1"**); *//设置k1的超时时间是100毫秒* **redisTemplate**.expire(**"k1"**,100, TimeUnit.***SECONDS***); *//设置k1为永久有效* **redisTemplate**.persist(**"k1"**); *//获取K1的超时时间* System.***out***.println(**redisTemplate**.getExpire(**"k1"**)); |

## 2.redis中的记录何时被回收？

如果是长期有效的==》 不回收

如果已过期的记录==》 默认情况下 也不回收，直到空间存满，只能读不能写。

默认什么情况下会回收呢？ ==》

set k1 v1

expire k1 30

===30秒到===

再次get k1 == 就回收了

## 3.redis中的常用内存管理算法有哪些

lru: Least recently used 最近最少使用

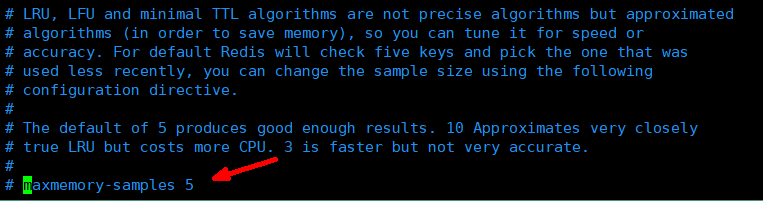
ttl 谁先挂谁先回收

random 随机

=====1.面向已超时的key，2.面向所有key====

noeviction 不回收

## 4.如何进行回收



1）精确回收

在取样时，取比较大的值，精确度高，但速度慢

2）非精确回收

在取样时，取比较小的值，精确度低，但速度快

# 七、Redis的持久化机制

就是把数据永久的保存。本来数据是在内存里面，默认情况下，当redis服务器关闭时，redis中的数据要被保存在磁盘文件中，那redis服务器会自动将内存数据保存在/data/dump.rdb文件中。

因此，在使用docker来创建redis时，要带着data数据卷，防止redis服务器删除，数据依然还在。

|  |
| --- |
| $ docker run -v /usr/local/docker/redis/conf/redis.conf:/usr/local/etc/redis/redis.conf -v /usr/local/docker/redis/data:/data --name myredis -p 6379:6379 redis redis-server /usr/local/etc/redis/redis.conf |

## 1.持久化机制之快照

每经过这个时间段，如果有这样的数据量的变化，那么就进行一次持久化。

save 900 1

save 300 10

save 60 10000

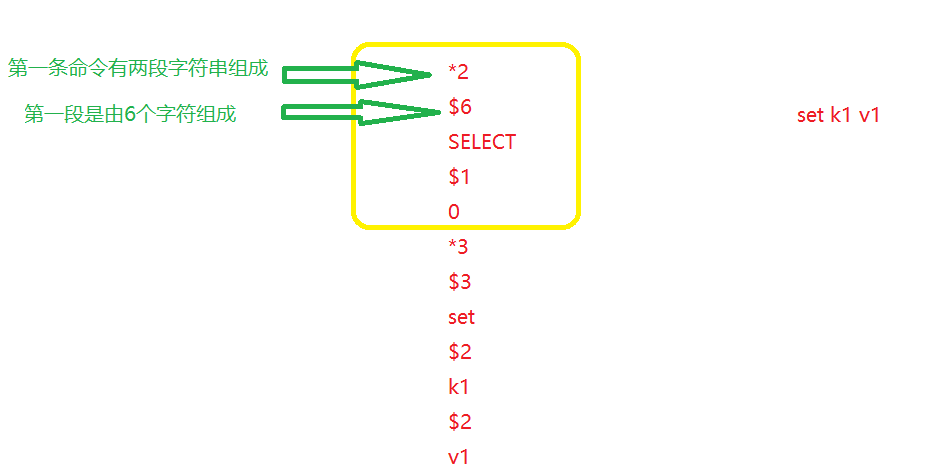
把数据存储在 /data/dump.rdb这个文件中

## 2.持久化机制之AOF(日志)

得通过配置文件appendonly no==> appendonly yes

会把执行的命令写在日志文件appendonly.aof当中

在写的过程当中，会把执行的命令通过一定的规律拆封成日志记录，存到文件中



那么，redis将命令写在日志里时是多少时间间隔做一次？

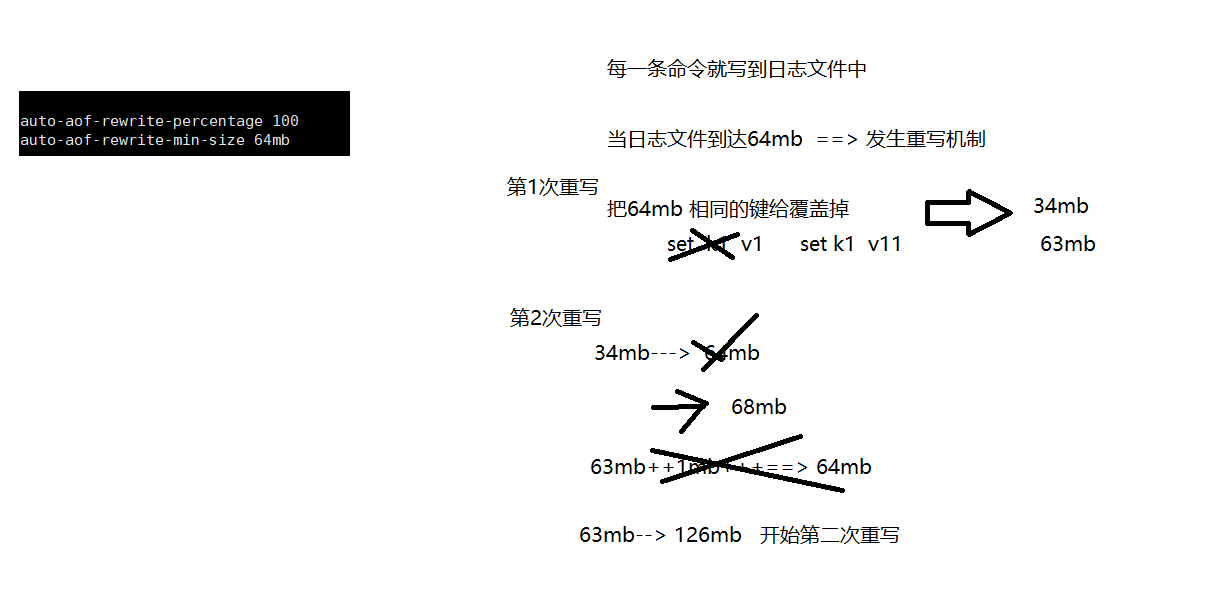
no fsync 不让redis操作，而是让操作系统去操作

always 每写一条语句，就写到日志里

everysec 每秒钟写一遍日志 《==比较好的方式，性能比较好！

日志文件并不会把重复的key覆盖，需要通过重写机制来实现日志文件的重复的key的删除。

- 关于什么时候重写？



- 如何重写？

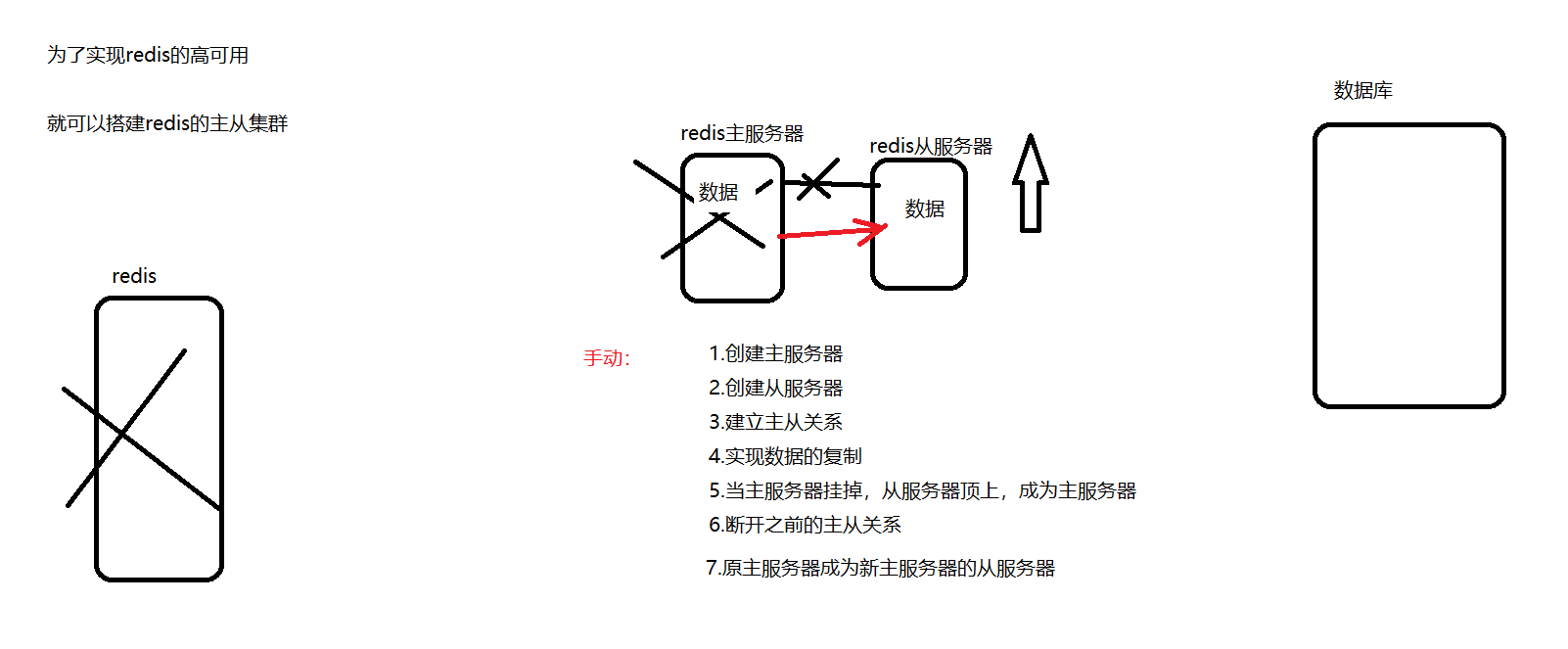
他将日志文件全部读到内存中，然后再把内存的数据写到日志文件里面。

# 八、Redis的高可用

## 1.如何实现高可用

高并发、高性能、高可用

所谓的高可用就是一直可以用，如何保证redis的高可用，通过搭建redis的主从服务器集群



linux中的文档的全局修改命令：

:%s/要修改的字符串/修改后的字符串

查看日志的linux命令： tail -f log.log 查看近一个月的日志的命令 sent 时间间隔

注意：

6380的从机使用redis-cli工具，必须指明6380端口，否则会使用默认的6379端口



./redis-cli -p 6380

## 2.实现步骤

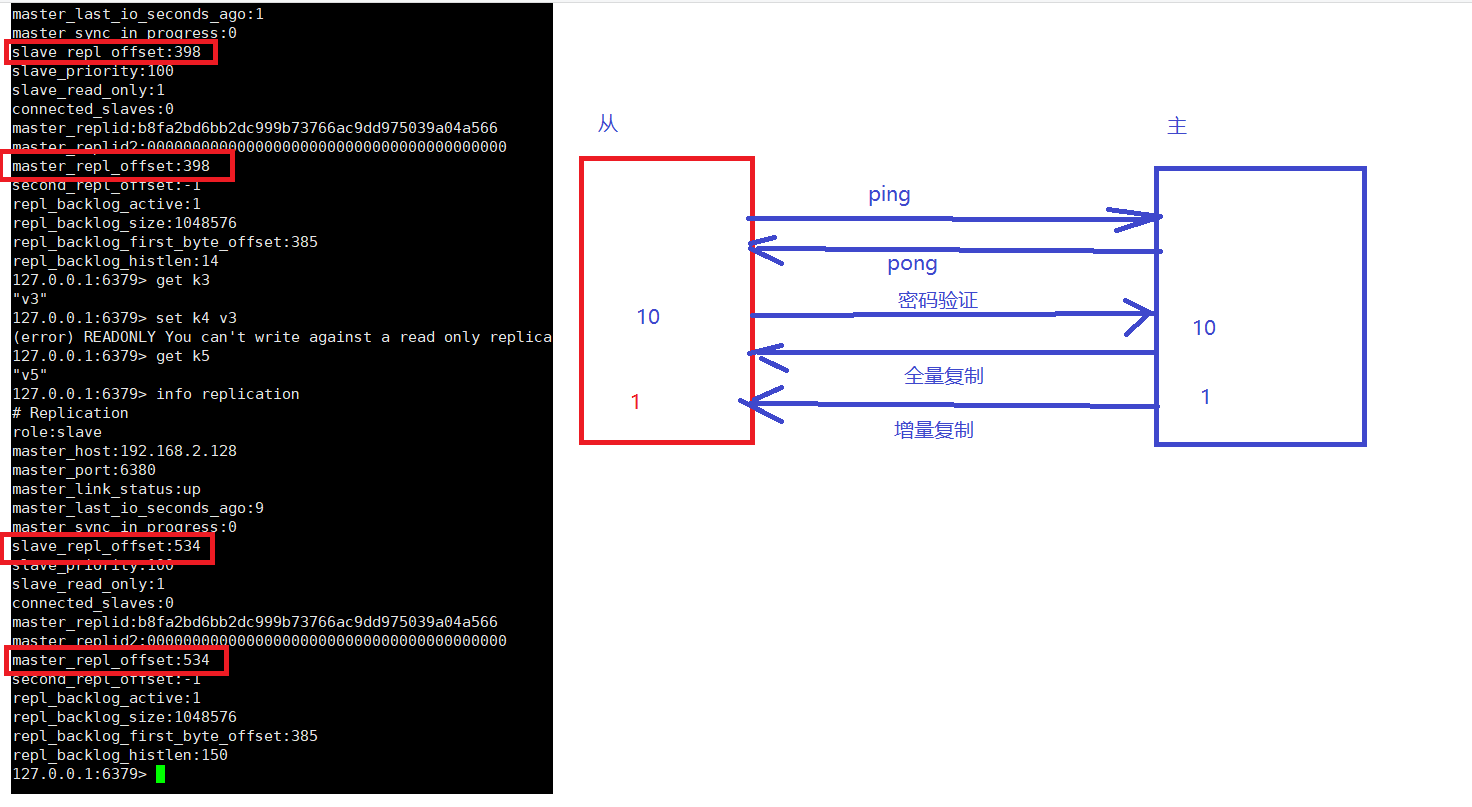
* 1搭建主服务器
* 2.搭建从服务器
* 3.建立主从关系，在从服务器上 使用： slaveof 主服务器ip 主服务器端口
* 4.实现数据复制
* 5.主服务器挂掉，从服务器上线，断开主从关系： slaveof no one
* 6.原主服务器成为新从服务器，又要建立主从关系

查看主从关系的命令： info replication

## 3.数据复制是如何实现的？

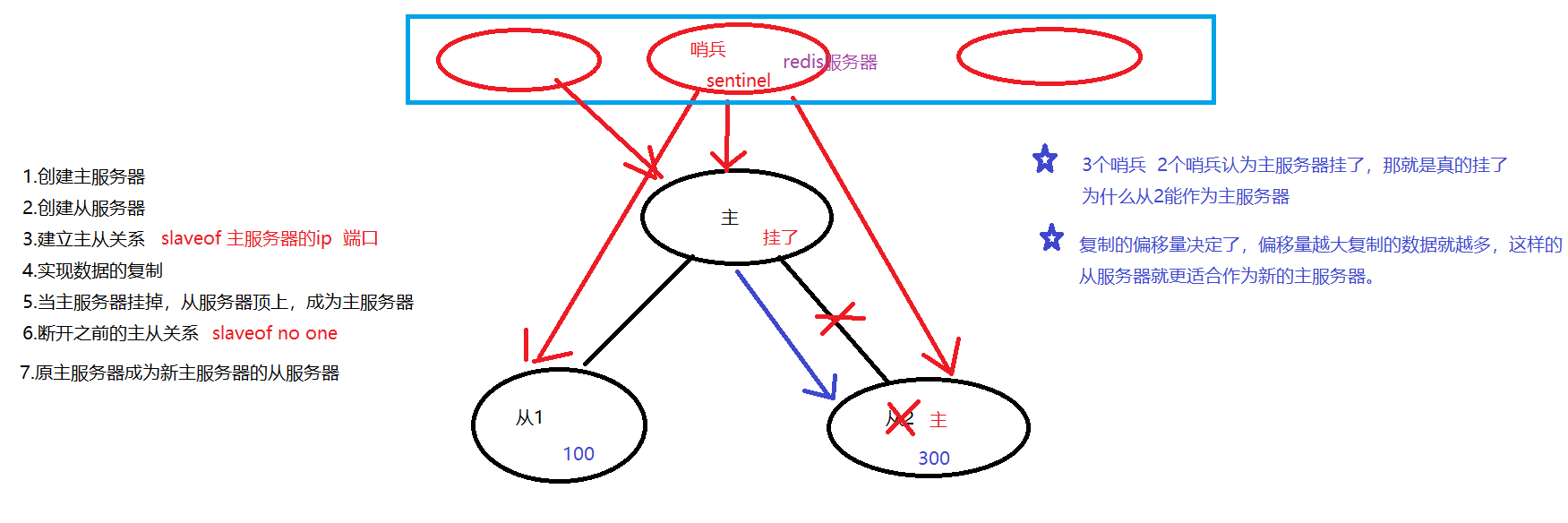
第一次是全量复制

之后是增量复制



## 4.redis-sentinel哨兵实现高可用

当主服务器挂了，从服务器顶上，重现建立主从关系，那么这些事情应该由程序自动完成。redis服务器可以充当哨兵身份来监控主服务器，当情况出现就出手！



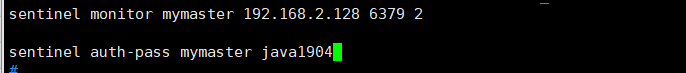
步骤：

1）搭建redis主从服务器，并建立主从关系

2）获得并修改配置文件： sentinel.conf

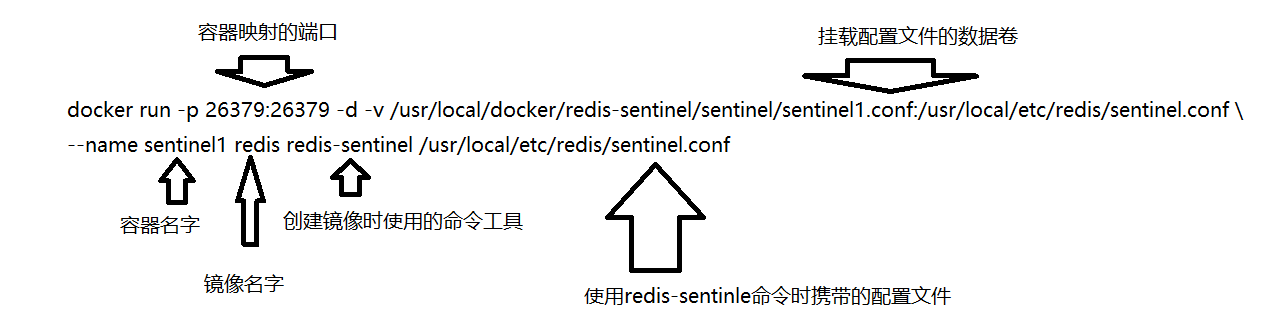
在里面修改三项：

* 当前sentinel的端口
* ip地址是redis主服务器(节点)的ip
* 密码是主节点的密码



3）复制得到三个sentinel.conf

4）使用docker来启动三个哨兵



## 5.spring-data-redis连接哨兵

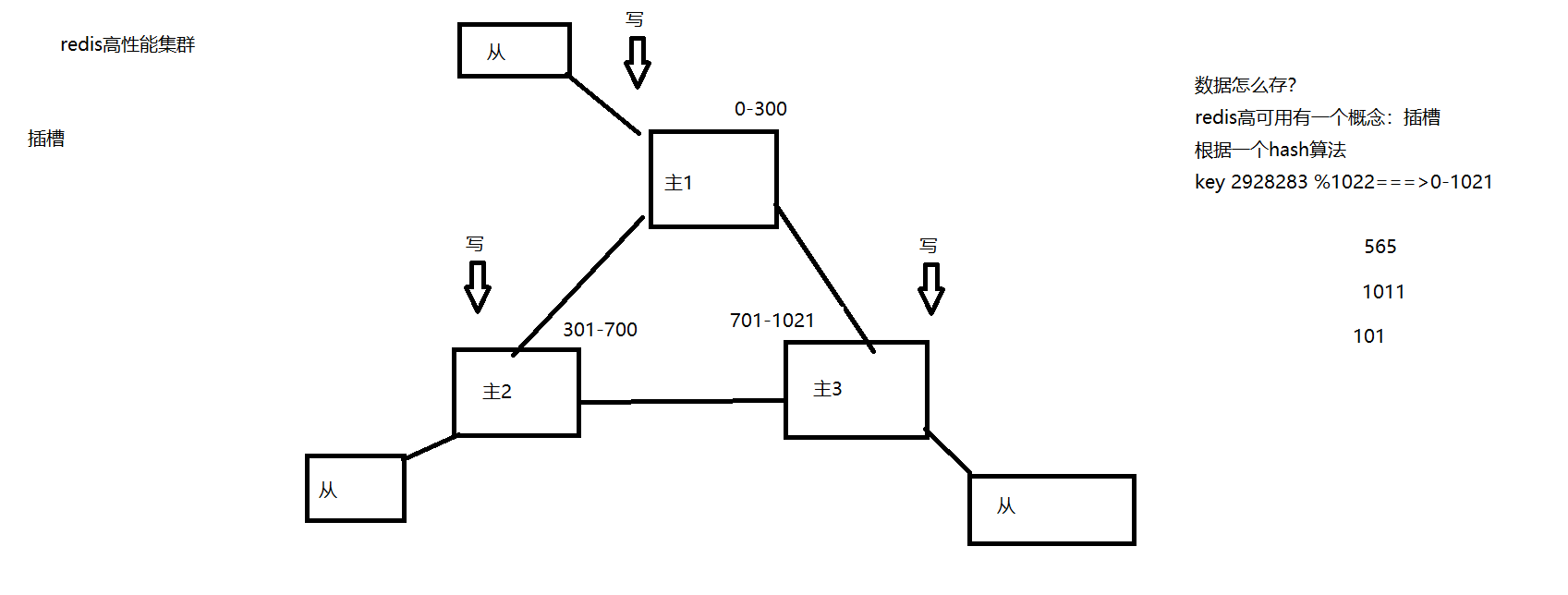
修改配置文件：

加入哨兵的配置

让连接工厂去连接哨兵，而非主节点

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd"**>    *<!--哨兵-->  <!-- 哨兵配置 -->* <**bean id="sentinelConfig" class="org.springframework.data.redis.connection.RedisSentinelConfiguration"**>  *<!-- 配置master主服务的名称 -->* <**property name="master"**>  <**bean class="org.springframework.data.redis.connection.RedisNode"**>  <**property name="name" value="mymaster"**/>  </**bean**>  </**property**>  *<!-- 配置哨兵服务 -->* <**property name="sentinels"**>  <**set**>  <**bean class="org.springframework.data.redis.connection.RedisNode"**>  <**constructor-arg name="host" value="192.168.2.128"**/>  <**constructor-arg name="port" value="26379"**/>  </**bean**>  <**bean class="org.springframework.data.redis.connection.RedisNode"**>  <**constructor-arg name="host" value="192.168.2.128"**/>  <**constructor-arg name="port" value="26380"**/>  </**bean**>  <**bean class="org.springframework.data.redis.connection.RedisNode"**>  <**constructor-arg name="host" value="192.168.2.128"**/>  <**constructor-arg name="port" value="26381"**/>  </**bean**>  </**set**>  </**property**>  </**bean**>     *<!-- 配置redis连接池对象 -->* <**bean id="poolConfig" class="redis.clients.jedis.JedisPoolConfig"**>  *<!-- 最大空闲数 -->* <**property name="maxIdle" value="50"** />  *<!-- 最大连接数 -->* <**property name="maxTotal" value="100"** />  *<!-- 最大等待时间 -->* <**property name="maxWaitMillis" value="20000"** />  </**bean**>   *<!-- 配置redis连接工厂 -->* <**bean id="connectionFactory"  class="org.springframework.data.redis.connection.jedis.JedisConnectionFactory"**>  *<!-- 连接池配置 -->* <**property name="poolConfig" ref="poolConfig"** />   *<!--连接哨兵-->* <**constructor-arg name="sentinelConfig" ref="sentinelConfig"**/>   *<!--&lt;!&ndash; 连接主机 &ndash;&gt;-->  <!--<property name="hostName" value="192.168.2.128" />-->  <!--&lt;!&ndash; 端口 &ndash;&gt;-->  <!--<property name="port" value="6379" />-->  <!-- 密码 -->* <**property name="password" value="java1904"** />  </**bean**>   <**bean id="stringRedisSerializer" class="org.springframework.data.redis.serializer.StringRedisSerializer"**> </**bean**>  <**bean id="jdkSerializationRedisSerializer" class="org.springframework.data.redis.serializer.JdkSerializationRedisSerializer"**/>       *<!-- 配置redis模板对象 -->* <**bean class="org.springframework.data.redis.core.RedisTemplate"**>  *<!-- 配置连接工厂 -->* <**property name="connectionFactory" ref="connectionFactory"** />  <**property name="keySerializer" ref="stringRedisSerializer"**/>  <**property name="valueSerializer" ref="stringRedisSerializer"**/>  <**property name="hashKeySerializer" ref="stringRedisSerializer"**/>  <**property name="hashValueSerializer" ref="stringRedisSerializer"**/>  </**bean**>    *<!--自己实现的监听器-->* <**bean id="myRedisListener" class="com.qf.listener.MyRedisListener"**></**bean**>   *<!-- 配置消息监听容器 -->* <**bean id="messageListenerContainer" class="org.springframework.data.redis.listener.RedisMessageListenerContainer"  destroy-method="destroy"**>  *<!-- redis连接工厂 -->* <**property name="connectionFactory" ref="connectionFactory"**/>  *<!-- 配置监听配置 -->* <**property name="messageListeners"**>  <**map**>  *<!-- 配置监听器 -->* <**entry key-ref="myRedisListener"**>  *<!-- 配置监听的渠道 -->* <**bean class="org.springframework.data.redis.listener.ChannelTopic"**>  *<!-- 配置要监听的渠道名字 -->* <**constructor-arg name="name" value="nba"**/>  </**bean**>  </**entry**>  </**map**>  </**property**>  </**bean**>     </**beans**> |

# 九、搭建Redis高性能集群



具体的搭建步骤：跟着文档一步步来即可。

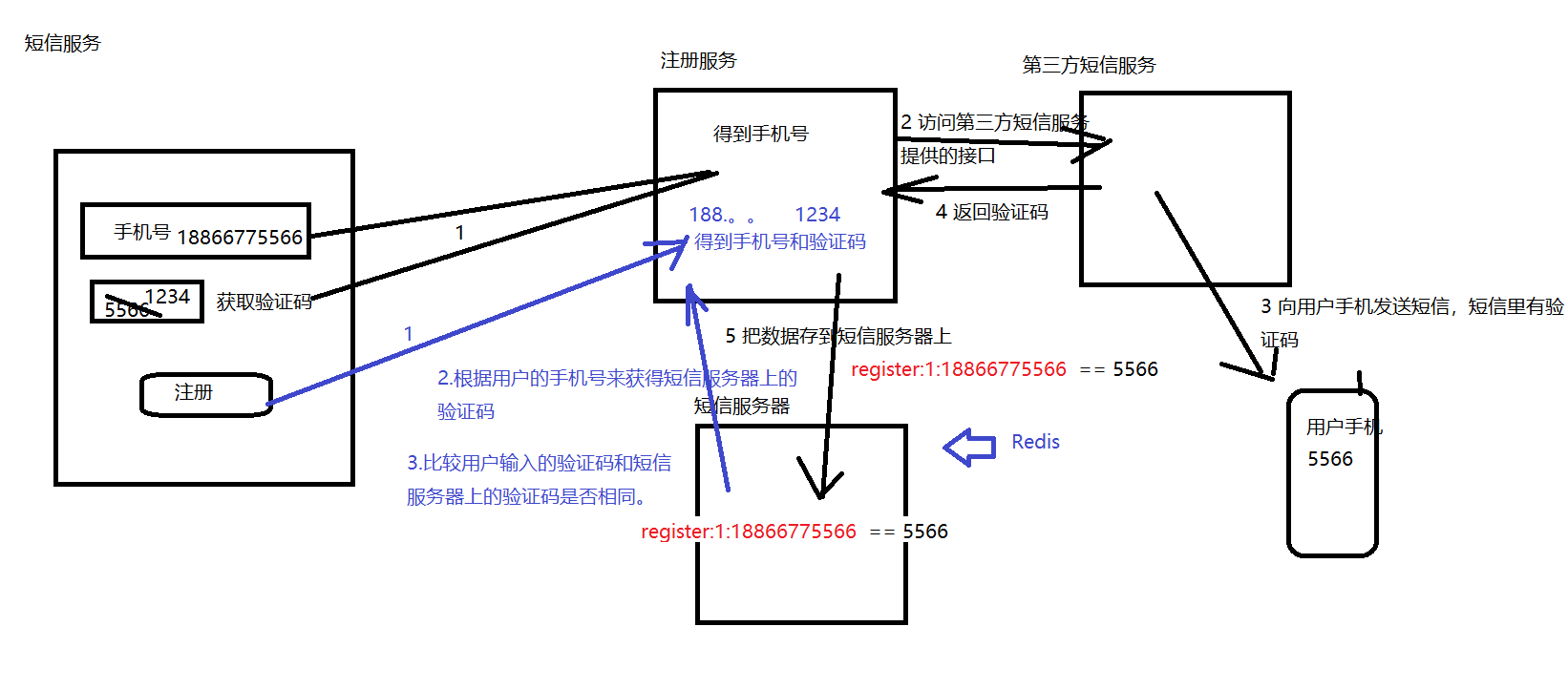
如何在spring-data-redis中使用redis高性能集群

修改配置文件

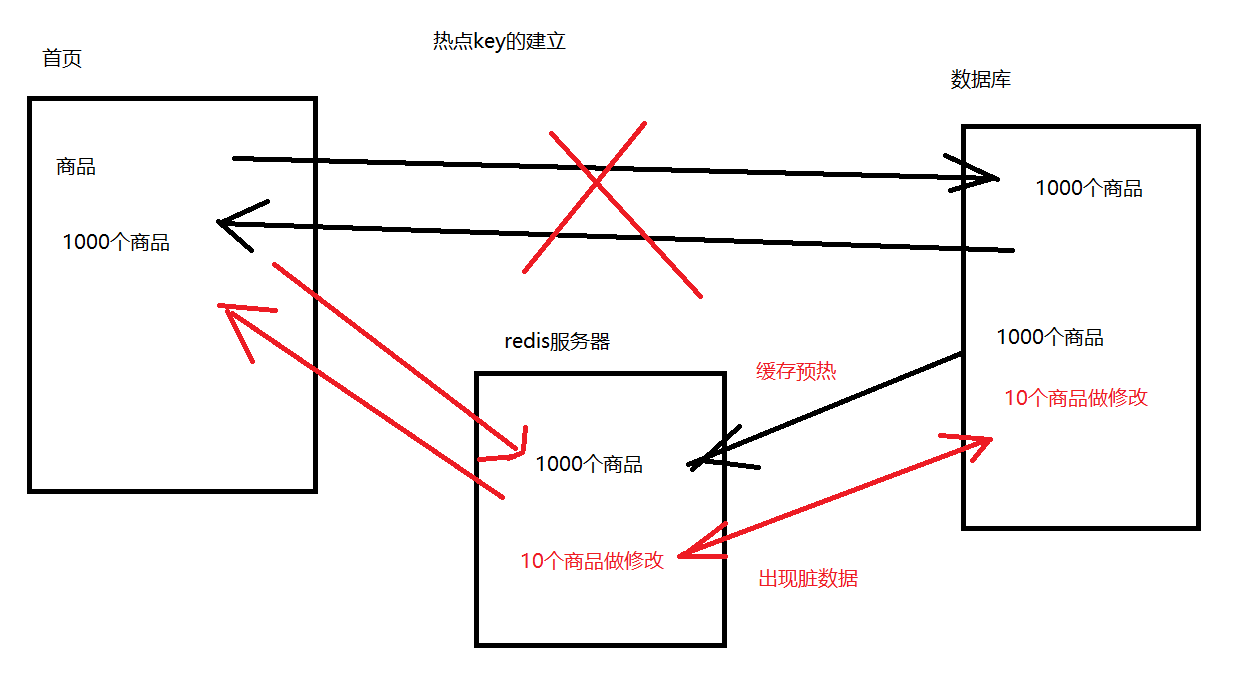
参考文档

# 十、Redis实战

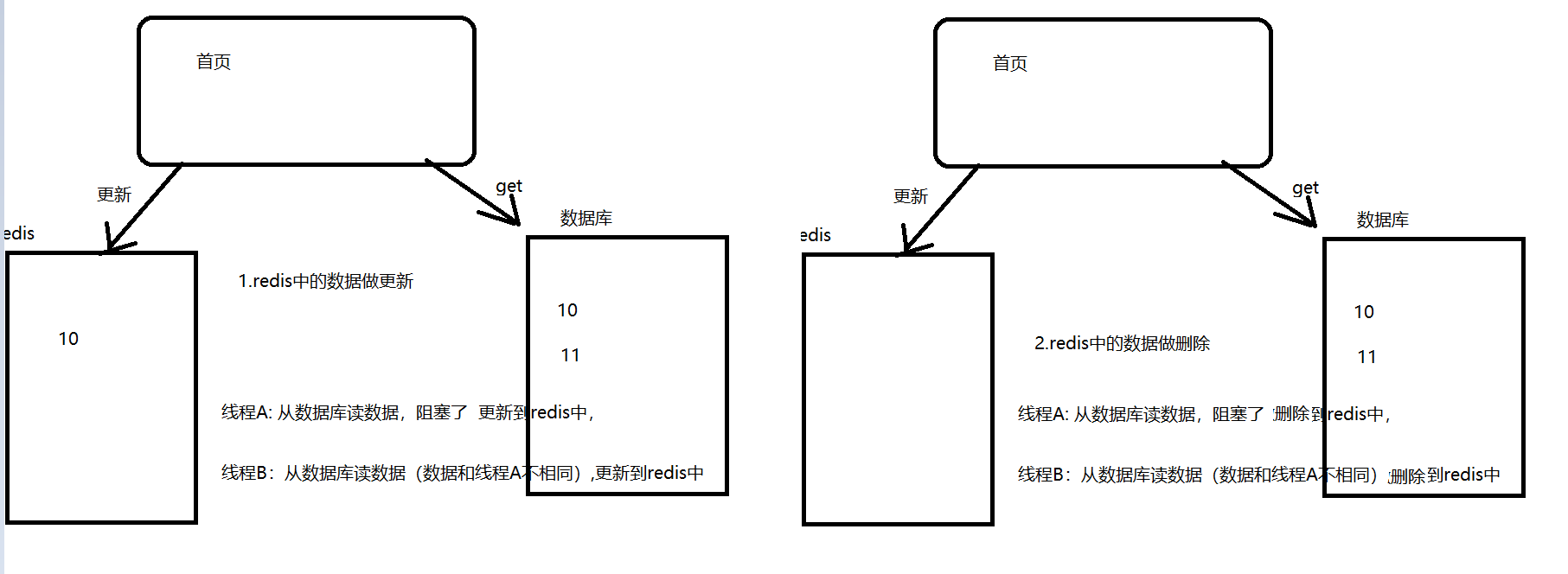
## 1.短信服务



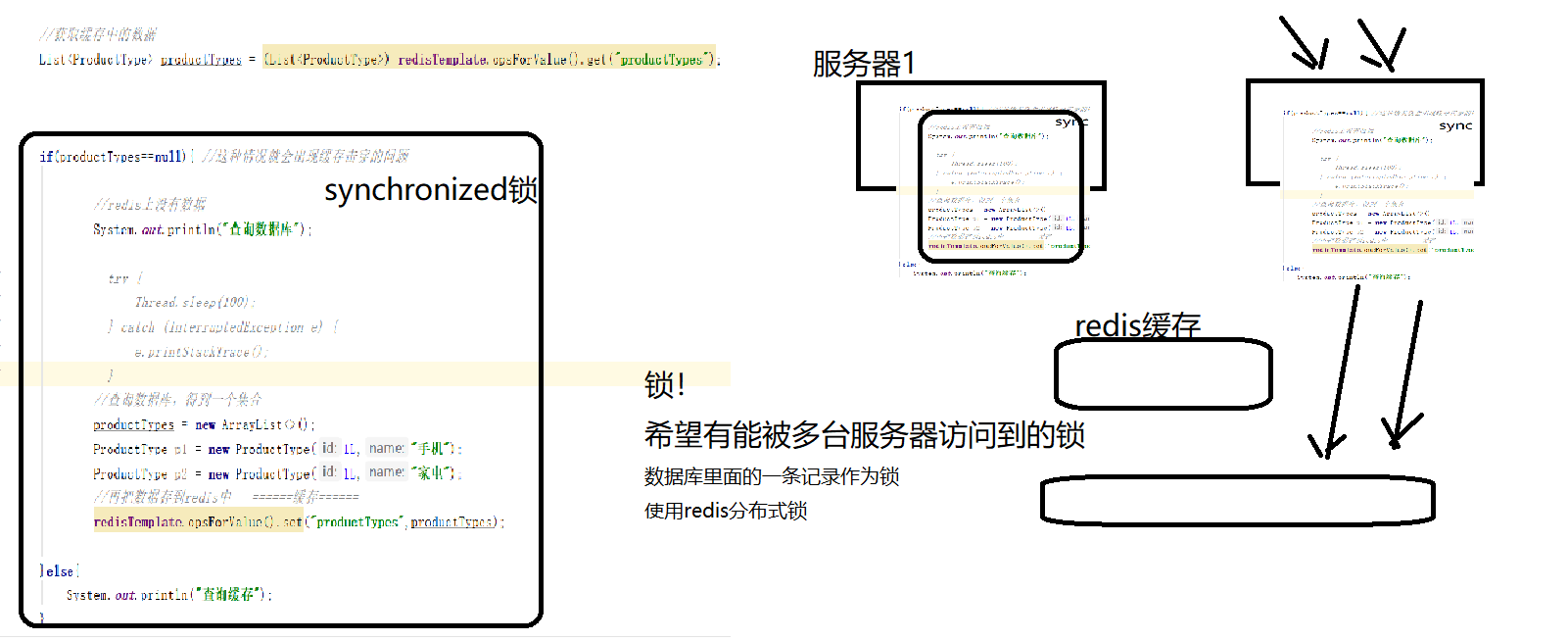
## 2.数据缓存



## 3.避免出现脏数据的更新策略



## 4.缓存击穿



避开了缓存，直接访问数据库，通过分布式锁的方式来解决缓存击穿问题。

解决方案：

1）使用分布式锁解决

|  |
| --- |
| *//redis实现数据缓存： 带分布式锁的方法，解决了缓存击穿的问题* @Test **public void** testRedisCacheData1(){   *//获取缓存中的数据* List<ProductType> productTypes = (List<ProductType>) **redisTemplate**.opsForValue().get(**"productTypes"**);     **if**(productTypes==**null**){ *//这种情况就会出现缓存击穿的问题  //setIfAbsent==> setnx* Boolean absent = **redisTemplate**.opsForValue().setIfAbsent(**"product:type:lock"**, 1);  *//absent==true 意味着成功得到了锁* **if**(absent){  *//redis上没有数据* System.***out***.println(**"查询数据库"**);   *//查询数据库，得到一个集合* productTypes = **new** ArrayList<>();  ProductType p1 = **new** ProductType(1L,**"手机"**);  ProductType p2 = **new** ProductType(1L,**"家电"**);  *//再把数据存到redis中 ======缓存======* **redisTemplate**.opsForValue().set(**"productTypes"**,productTypes);  *//释放锁* **redisTemplate**.delete(**"product:type:lock"**);  }**else**{   **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  e.printStackTrace();  }   testRedisCacheData1();   }   }**else**{  System.***out***.println(**"查询缓存"**);  }  } |

2）解决死锁问题

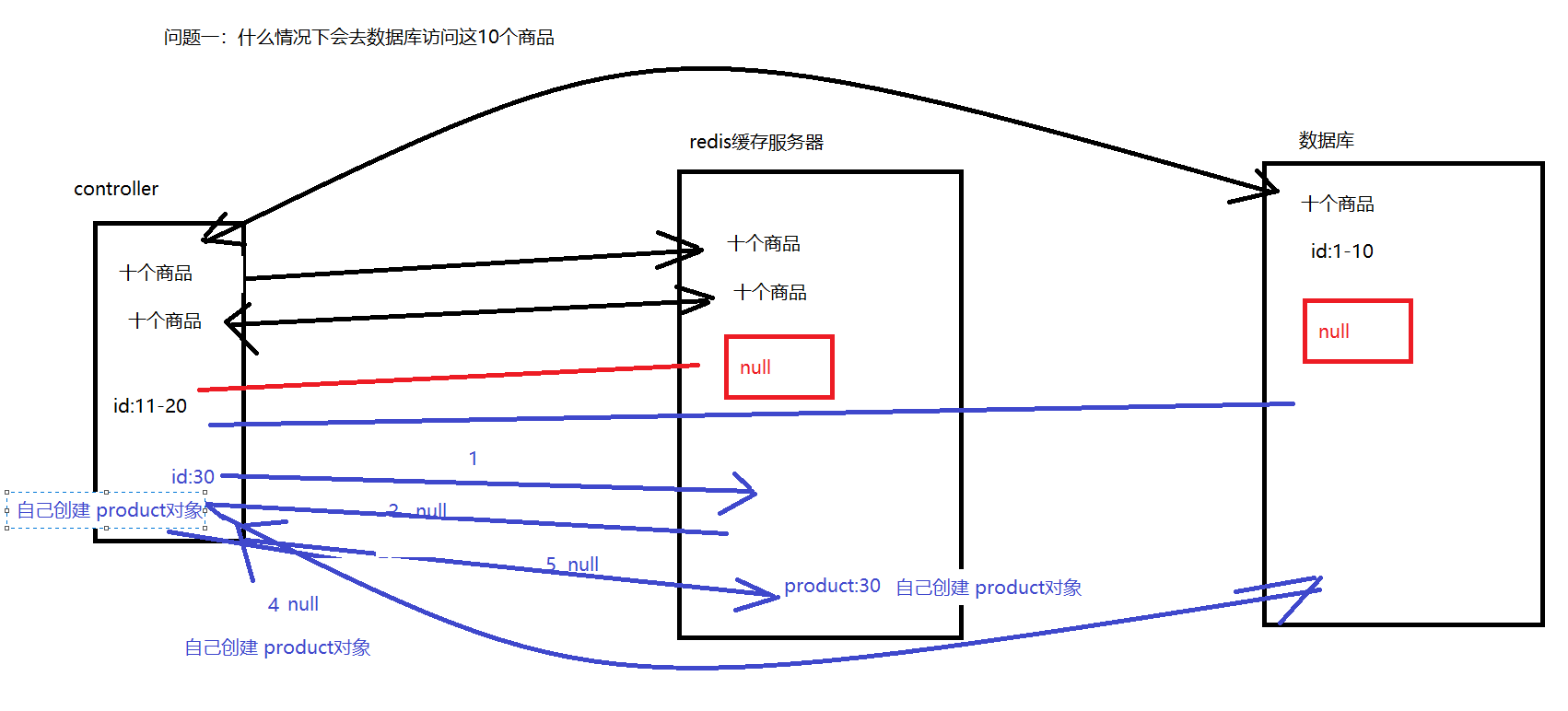
|  |
| --- |
| *//redis实现数据缓存： 带分布式锁的方法，解决了缓存击穿的问题 //死锁问题：方案一： try-finally 方案二： 设置锁的有效期，*  *//还要防止误删锁，如何解决误删锁，通过给锁设置值（uuid）,在删除锁之前，通过uuid进行判断该锁是否是自己的锁*  @Test **public void** testRedisCacheData2(){   *//获取缓存中的数据* List<ProductType> productTypes = (List<ProductType>) **redisTemplate**.opsForValue().get(**"productTypes"**);     **if**(productTypes==**null**){ *//这种情况就会出现缓存击穿的问题  //setIfAbsent==> setnx* String uuid = UUID.*randomUUID*().toString();  Boolean absent = **redisTemplate**.opsForValue().setIfAbsent(**"product:type:lock"**, uuid);   *//absent==true 意味着成功得到了锁* **if**(absent){  *//给锁设置超时时间。为了避免死锁问题,但是目前这种情况会造成误删锁。* **redisTemplate**.expire(**"product:type:lock"**,1000,TimeUnit.***MILLISECONDS***);  **try** {  *//redis上没有数据* System.***out***.println(**"查询数据库"**);  **int** i=10/0;  *//查询数据库，得到一个集合* productTypes = **new** ArrayList<>();  ProductType p1 = **new** ProductType(1L,**"手机"**);  ProductType p2 = **new** ProductType(1L,**"家电"**);  *//再把数据存到redis中 ======缓存======* **redisTemplate**.opsForValue().set(**"productTypes"**,productTypes);  *//如果超过了1000毫秒* } **finally** {  *//死锁问题的解决方案一： try-catch  //死锁问题的解决方案二： 设置锁的超时时间  //释放锁* String lock\_uuid = (String) **redisTemplate**.opsForValue().get(**"product:type:lock"**);  **if**(uuid.equals(lock\_uuid)){  *//防止误删锁，所以在删除锁之前，要看一下当前这个锁是不是自己创建的锁* **redisTemplate**.delete(**"product:type:lock"**);  }   }  }**else**{   **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  e.printStackTrace();  }   testRedisCacheData1();   }   }**else**{  System.***out***.println(**"查询缓存"**);  }  } |

## 5.缓存穿透

用户故意去差一个数据库中没有的数据，所以即使该数据会被存入到redis服务器中，依然是一个null值的数据，那么就会导致下次再访问这个数据时 ，依然会去查询数据库，因为redis服务器返回的是一个null值的数据。于是就会造成用户利用这一特性，每次都能直接访问数据库。——缓存穿透

缓存穿透的解决方案：

1）简单粗暴： 用户去数据库查到的数据是null，按照原来的思路 往缓存里存的也是null，但是可以设置，如果数据库查到的是null（数据库里没有这条记录的意思）,那么就创建一个临时对象，存入到缓存中，防止用于下次再访问该对象时缓存中没有而直接去访问数据库。并且在缓存中给该对象设置一个较短的有效期（一般为五分钟）。



|  |
| --- |
| *//模拟去数据库根据id获得商品* **public** Product getProductById(Long id) {   **if** (id <= 10) {  Product p = **new** Product(id, **""** + id);  **return** p;  } **else** {  **return null**;  }   }   @Test **public void** testCacheBroke1() {   **for** (**int** i = 11; i <= 20; i++) {  *//去缓存里拿id是11-120的十个商品* Product p = (Product) **redisTemplate**.opsForValue().get(**"product:"** + i);  **if** (p == **null**) {  System.***out***.println(**"查询数据库"**);  *//发现缓存里没有  //去数据库拿* Product product = getProductById((**long**) i);*//11-20 ==> null* **if**(product==**null**){  *//说明数据库里根本就没有这样的数据* product = **new** Product(9999999L,**""**);  **redisTemplate**.opsForValue().set(**"product:"** + i, product);  *//给这个临时的键设一个较短的有效期* **redisTemplate**.expire(**"product:"** + i,10,TimeUnit.***MILLISECONDS***);  }**else**{  *//把数据存入到缓存中* **redisTemplate**.opsForValue().set(**"product:"** + i, product);  }    } **else** {  System.***out***.println(**"查询缓存"**);  }  } } |

解决方案二：布隆过滤器

