# Redis实战原理课程介绍

## Redis缓存框架基本介绍

Redis 是完全开源免费的，是一个高性能的key-value数据库，目前市面上主流的数据库

Redis、Memcache、Tair(淘宝自研发)

Redis的官网：<https://redis.io/>

内存数据库（nosql数据库）、mysql、sqlserver

关系数据库存放在硬盘中 查询实现io操作

非关系数据库 Redis 持久化机制 淘汰策略 （分布式共享）

Jvm内置缓存框架 ECACH os cache （1个JVM缓存修改后需要各个节点进行同步）

## Redis的应用场景

1. Token令牌的生成
2. 短信验证码Code
3. 缓存查询数据
4. 网页计数器
5. 分布式锁
6. 延迟操作

# Redis环境安装linux

注意：Redis官方是没有windows版本的，只有linux，这时候因为 在nio中epoll只有linux操作系统独有

# Redis单线线程模型

参考第6期第62节

首先Redis官方是没有windows版本的，只有redis版本

Redis的底层采用Nio中的多路IO复用的机制，能够非常好的支持这样的并发，从而保证线程安全问题；

Redis单线程，也就是底层采用一个线程维护多个不同的客户端io操作。

但是Nio在不同的操作系统上实现的方式有所不同，在我们windows操作系统使用select实现轮训时间复杂度是为o(n)，而且还存在空轮训的情况，效率非常低， 其次是默认对我们轮训的数据有一定限制，所以支持上万的tcp连接是非常难。

所以在linux操作系统采用epoll实现事件驱动回调，不会存在空轮训的情况，只对活跃的 socket连接实现主动回调这样在性能上有大大的提升，所以时间复杂度是为o(1)

注意：windows操作系统是没有epoll，只有linux系统才有epoll

所以为什么nginx、redis都能够非常高支持高并发，最终都是linux中的IO多路复用机制epoll

Redis底层采用nio epoll实现

## Linux环境下安装Redis

### 环境安装Redis

|  |
| --- |
| 1. 上传Redis的安装包   redis-5.0.6.tar.gz   1. 解压我们的Redis安装包   tar -zxvf redis-5.0.6.tar.gz  进入目录  cd redis-5.0.6/   1. 安装GCC环境   yum install gcc  yum install gcc-c++  make MALLOC=libc   1. mkdir /usr/redis 2. make install PREFIX=/usr/redis 3. 启动Redis cd /usr/redis/bin ./redis-server |
|  |

### 环境核心配置

#### 将Redis设置为后台启动

cp /usr/redis-5.0.6/redis.conf /usr/redis/bin

vi redis.conf daemonize yes

./redis-server ./redis.conf 重启启动Redis

ps aux | grep 'redis'

启动redis客户端

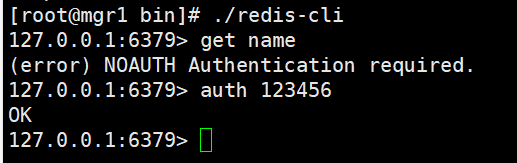
./redis-cli

#### 设置Redis账号密码

搜索redis.conf：# requirepass foobared

requirepass 123456

客户端连接：auth 123456



#### 设置Reids允许ip访问

先关闭防火墙

Systemctl stop firewalld

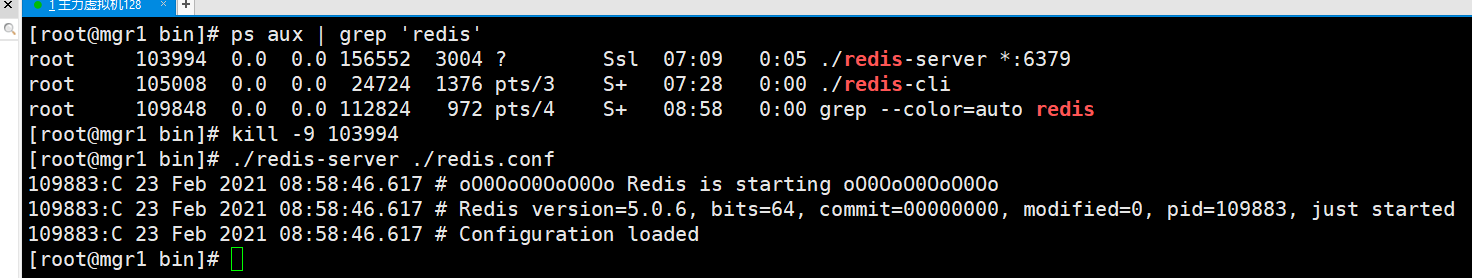
使用Redis客户端连接后发现连接失败，需要修改配置文件允许外部访问



修改redis.conf注释掉bind 192.168.65.128

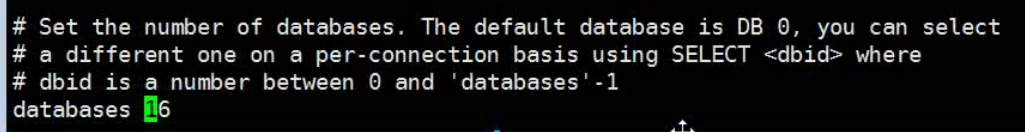
**protected-mode no ###允许外界访问**

**重启redis服务**



#### Redis默认的情况下分为16个库

**0-15**



## Redis数据结构

String类型、Hsh类型、List类型、Set类型 、Sorted-Sets

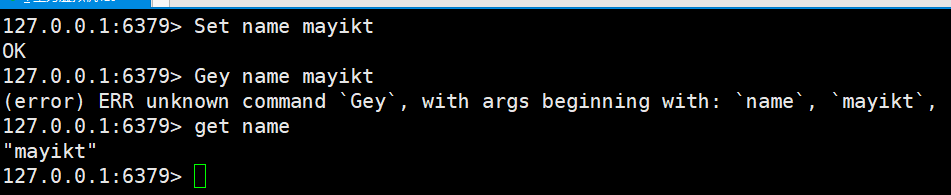
./redis-cli -h 192.168.65.128 -p 6379 -a 123456

### String类型

String是redis最基本的类型，一个key对应一个value，sring类型是二进制安全的。意思是redis的string可以包含任何数据。比如jpg图片或者序列化的对象, Sring类型是Redis最基本的数据类型，一个键最大能存储512MB。

Set name mayikt

Gey name mayikt

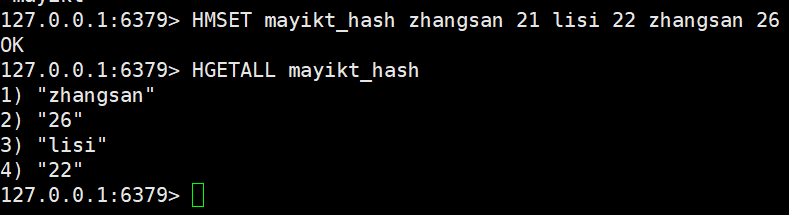


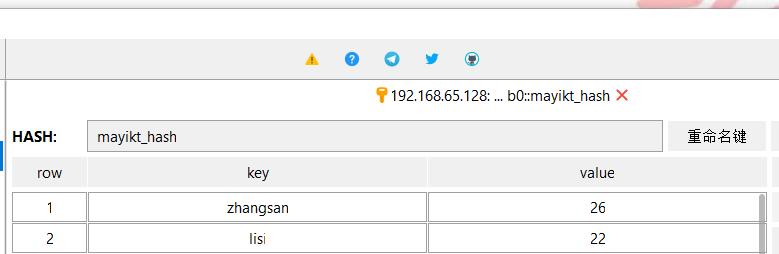
### Hash类型

我们可以将Redis中的Hash类型看成具有<key,<key1,value>>,其中同一个key可以有多个不同key值的<key1,value>，所以该类型非常适合于存储值对象的信息。如Username、Password和Age等。如果Hash中包含很少的字段，那么该类型的数据也将仅占用很少的磁盘空间。

HMSET mayikt\_hash zhangsan 21 lisi 22 zhangsan 26

HGETALL mayikt\_hash



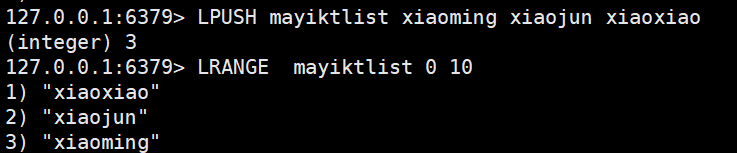


### List类型

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）

LPUSH mayiktlist xiaoming xiaojun xiaoxiao

LRANGE mayiktlist 0 10



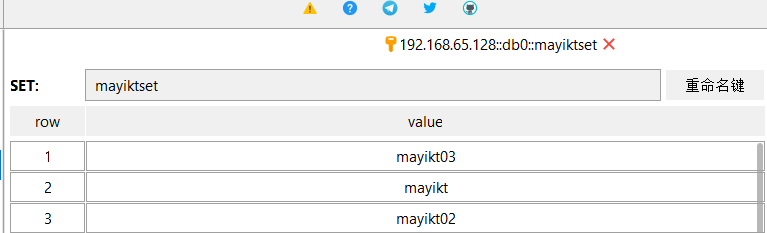
### Redis 集合(Set)

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)。

SADD mayiktset mayikt mayikt02 mayikt03 mayikt03 mayikt03

SMEMBERS mayiktset



### 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)。

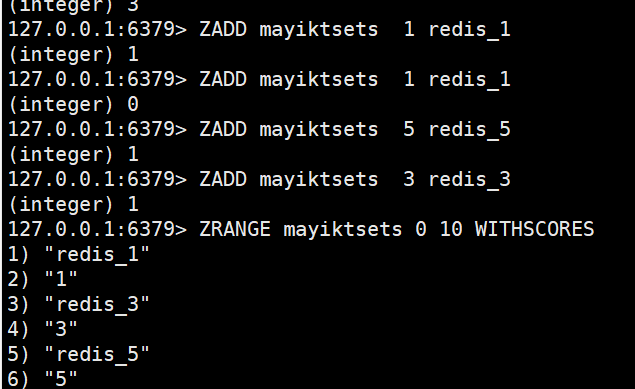
ZADD mayiktsets 1 redis\_1

ZADD mayiktsets 1 redis\_1

ZADD mayiktsets 5 redis\_5

ZADD mayiktsets 3 redis\_3

ZRANGE mayiktsets 0 10 WITHSCORES



Redis如何存放一个java对象 直接存放json类型即可

Set key=user value=｛”userid”:100,”username”:yushengjun｝

Xxl-sso 底层如何存放对象 存放二进制

# SpringBoot整合Redis

**项目：mayikt\_redis\_course01\_01\_Redis整合SpringBoot**

方案1：在Redis存放一个对象 使用json序列化与反序列化

方案2：直接使用redis自带序列化方式存储对象

## 方案1存储json

### Maven依赖

|  |
| --- |
| <**parent**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-parent</**artifactId**>  <**version**>2.0.1.RELEASE</**version**> </**parent**> <**dependencies**>  *<!-- 集成commons工具类 -->* <**dependency**>  <**groupId**>org.apache.commons</**groupId**>  <**artifactId**>commons-lang3</**artifactId**>  </**dependency**>  *<!-- 集成lombok 框架 -->* <**dependency**>  <**groupId**>org.projectlombok</**groupId**>  <**artifactId**>lombok</**artifactId**>  </**dependency**>  *<!-- fastjson -->* <**dependency**>  <**groupId**>com.alibaba</**groupId**>  <**artifactId**>fastjson</**artifactId**>  <**version**>1.2.30</**version**>  </**dependency**>  *<!-- SpringBoot-整合Web组件 -->* <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-web</**artifactId**>  </**dependency**>  <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-data-redis</**artifactId**>  </**dependency**> </**dependencies**> *<!-- 管理依赖 -->* <**dependencyManagement**>  <**dependencies**>  <**dependency**>  <**groupId**>org.springframework.cloud</**groupId**>  <**artifactId**>spring-cloud-dependencies</**artifactId**>  <**version**>Finchley.M7</**version**>  <**type**>pom</**type**>  <**scope**>import</**scope**>  </**dependency**>  </**dependencies**> </**dependencyManagement**> |

### 工具类

|  |
| --- |
| @Component **public class** RedisUtils {   @Autowired  **private** StringRedisTemplate **stringRedisTemplate**;   **public void** setString(String key, String value) {  setString(key, value, **null**);  }   **public void** setString(String key, String value, Long timeOut) {  **stringRedisTemplate**.opsForValue().set(key, value);  **if** (timeOut != **null**) {  **stringRedisTemplate**.expire(key, timeOut, TimeUnit.***SECONDS***);  }  }   **public** String getString(String key) {  **return stringRedisTemplate**.opsForValue().get(key);  } } |

### 控制层

|  |
| --- |
| @RestController public class RedisController {    @Autowired  private RedisUtils redisUtils;   @GetMapping("/addUser")  public String addUser(UserEntity userEntity) {  // 1.将我们的对象转换成json  String json = JSONObject.*toJSONString*(userEntity);  redisUtils.setString("userEntity", json);  return "存储成功~";  }   @RequestMapping("/getUser")  public UserEntity getUser(String key) {  String json = redisUtils.getString(key);  UserEntity userEntity = JSONObject.*parseObject*(json, UserEntity.class);  return userEntity;  } } |

### 配置文件

|  |
| --- |
| spring:  redis:  host: 192.168.65.128  password: 123456  port: 6379  database: 1 |

### 运行测试

<http://localhost:8080/addUser?userId=122&userName=caijingwen>



## 方案2序列化方式

### 工具类

|  |
| --- |
| @Component **public class** RedisTemplateUtils {   @Resource  **private** RedisTemplate<String, Object> **redisTemplate**;   **public void** setObject(String key, Object object) {  **redisTemplate**.opsForValue().set(key, object);  }   **public** Object getObjet(String key) {  **return redisTemplate**.opsForValue().get(key);  }  } |

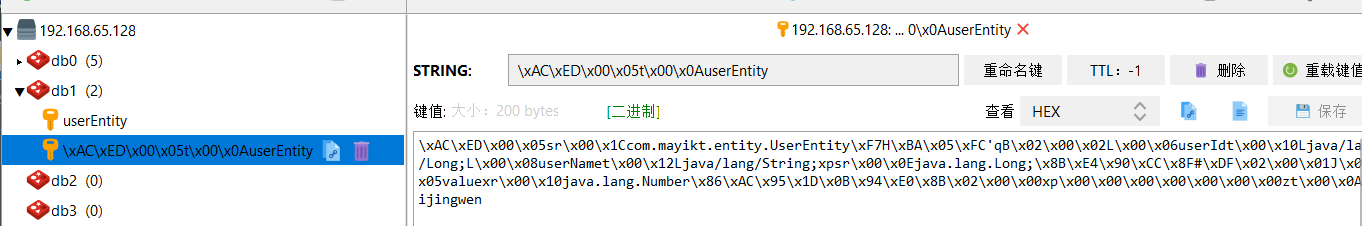
### 实体类

|  |
| --- |
| @Data **public class** UserEntity **implements** Serializable {  **private** Long **userId**;  **private** String **userName**; } |

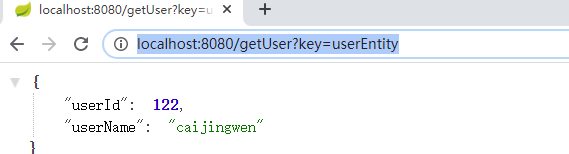
注意需要序列化的对象一定要实现Serializable接口

### 运行

http://localhost:8080/addUser?userId=122&userName=caijingwen



http://localhost:8080/getUser?key=userEntity



# MySQL与Redis一致性解决同步问题

**项目：mayikt\_redis\_course01\_02\_SpringBoot Redis注解@Cached**

SpringBoot整合Redis的注解版本

|  |
| --- |
| @RestController **public class** MemberController {  @Autowired  **private** UserMapper **userMapper**;   @RequestMapping(**"/findMemberAll"**)  @Cacheable(cacheNames = **"member"**, key = **"'findMemberAll'"**)  **public** List<MemberEntity> findMemberAll() {  **return userMapper**.findMemberAll();  } } |

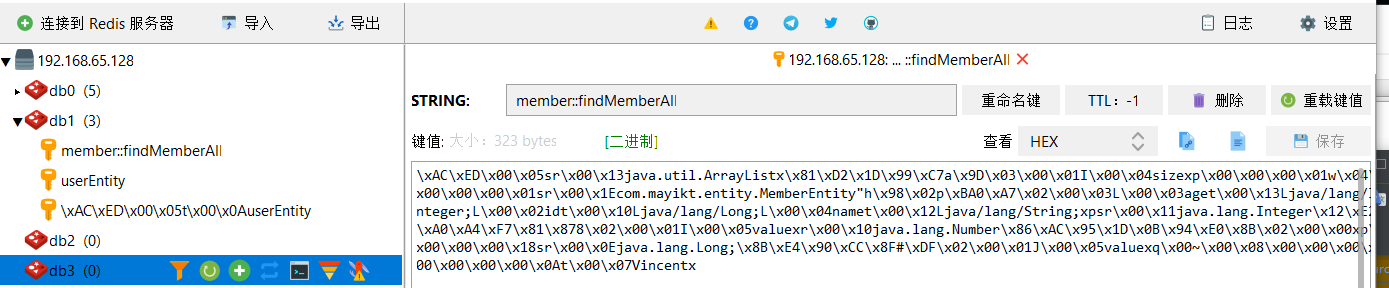
|  |
| --- |
| **public interface** UserMapper {  @Select(**"select \* from users"**)  List<MemberEntity> findMemberAll(); } |

|  |
| --- |
| **spring**:  **redis**:  **host**: 192.168.212.155  **password**: 123456  **port**: 6379  **database**: 1  *#数据库连接配置* **datasource**:  **driver-class-name**: com.mysql.jdbc.Driver  **url**: jdbc:mysql://192.168.212.163:3306/user?characterEncoding=utf-8&useSSL=false  **username**: root  **password**: root |

|  |
| --- |
| *<!--mysql数据库驱动-->* <**dependency**>  <**groupId**>mysql</**groupId**>  <**artifactId**>mysql-connector-java</**artifactId**> </**dependency**>  *<!--mybatis-->* <**dependency**>  <**groupId**>org.mybatis.spring.boot</**groupId**>  <**artifactId**>mybatis-spring-boot-starter</**artifactId**>  <**version**>2.1.0</**version**> </**dependency**> |

|  |
| --- |
| 开启我们的@EnableCaching |

http://localhost:8080/findMemberAll



MySQL与Redis一致性解决同步问题

方式1：直接清除Redis的缓存，重新读取数据库即可

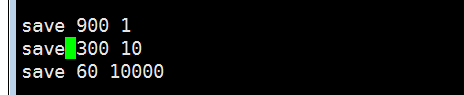
方式2：使用mq异步订阅mysql binlog实现增量同步

方式3：使用alibaba的canal（实现为方式2）

# Redis的RDB与AOF同步配置

### RDB

Redis默认采用rdb方式实现数据的持久化，以快照的形式将数据持久化到磁盘的是一个二进制的文件dump.rdb， 在redis.conf文件中搜索“dump.rdb “。



Redis会将数据集的快照dump到dump.rdb文件中。此外，我们也可以通过配置文件来修改Redis服务器dump快照的频率，在打开6379.conf文件之后，我们搜索save，可以看到下面的配置信息：

save 900 1 #在900秒(15分钟)之后，如果至少有1个key发生变化，则dump内存快照。

save 300 10 #在300秒(5分钟)之后，如果至少有10个key发生变化，则dump内存快照。

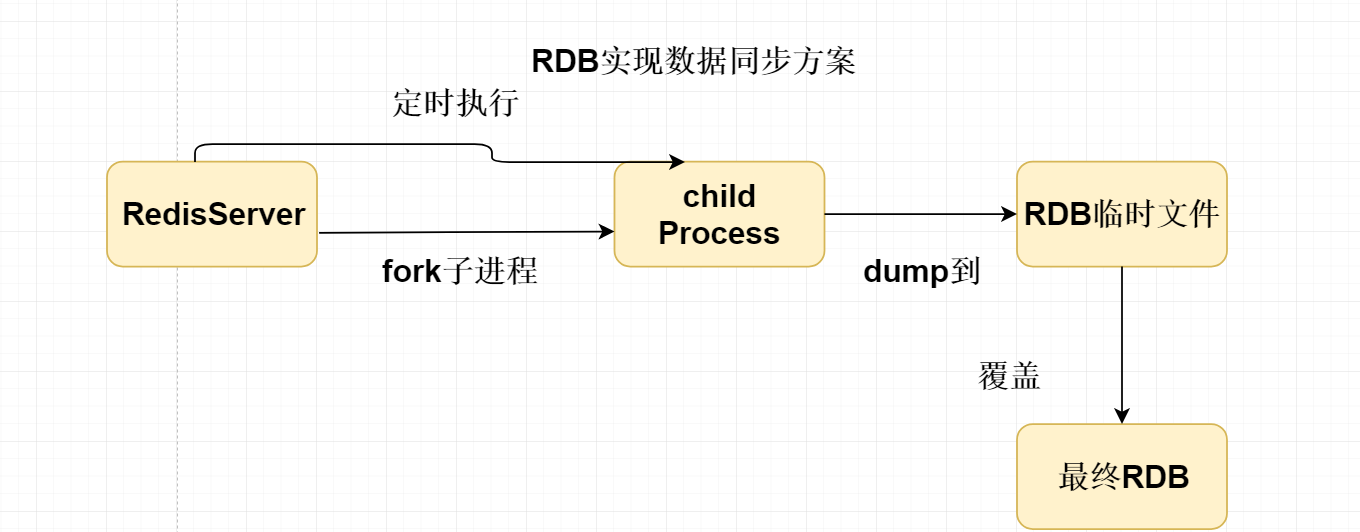
save 60 10000 #在60秒(1分钟)之后，如果至少有10000个key发生变化，则dump内存快照。

Set(包含增加和覆盖)、del

Set name yushengjun

Set name mayikt

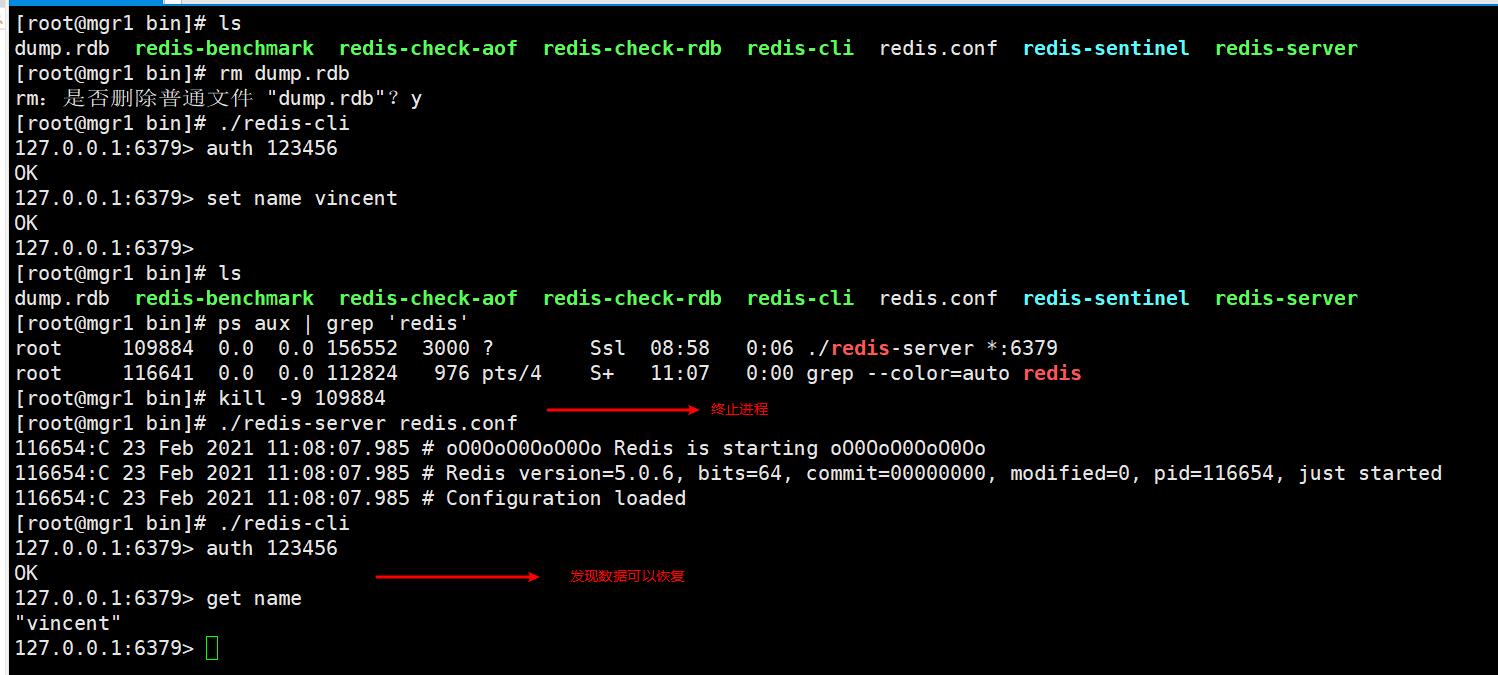
Del name



#### 测试

修改保存时间

save 900 1 改为 save 10 1



### Aof

在Redis的配置文件中存在三种同步方式，它们分别是：

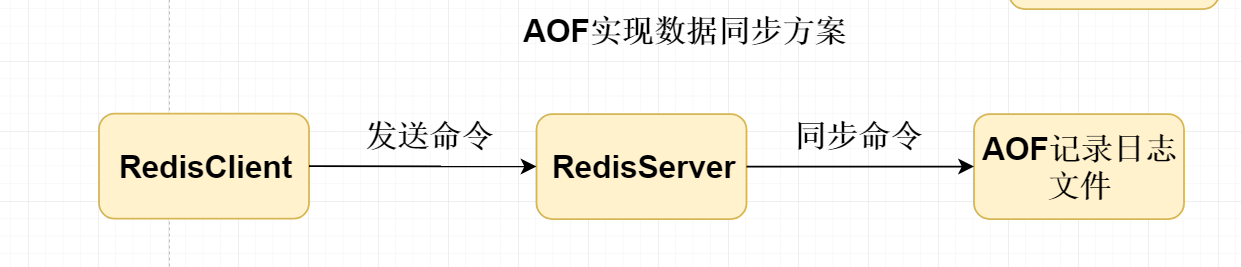
appendfsync always #每次有数据修改发生时都会写入AOF文件，能够保证数据不丢失，但是效率非常低。

appendfsync everysec #每秒钟同步一次，可能会丢失1s内的数据，但是效率非常高。

appendfsync no #什么时候将缓冲区内容同步到AOF文件中，由操作系统决定

直接修改redis.conf中 appendonly yes

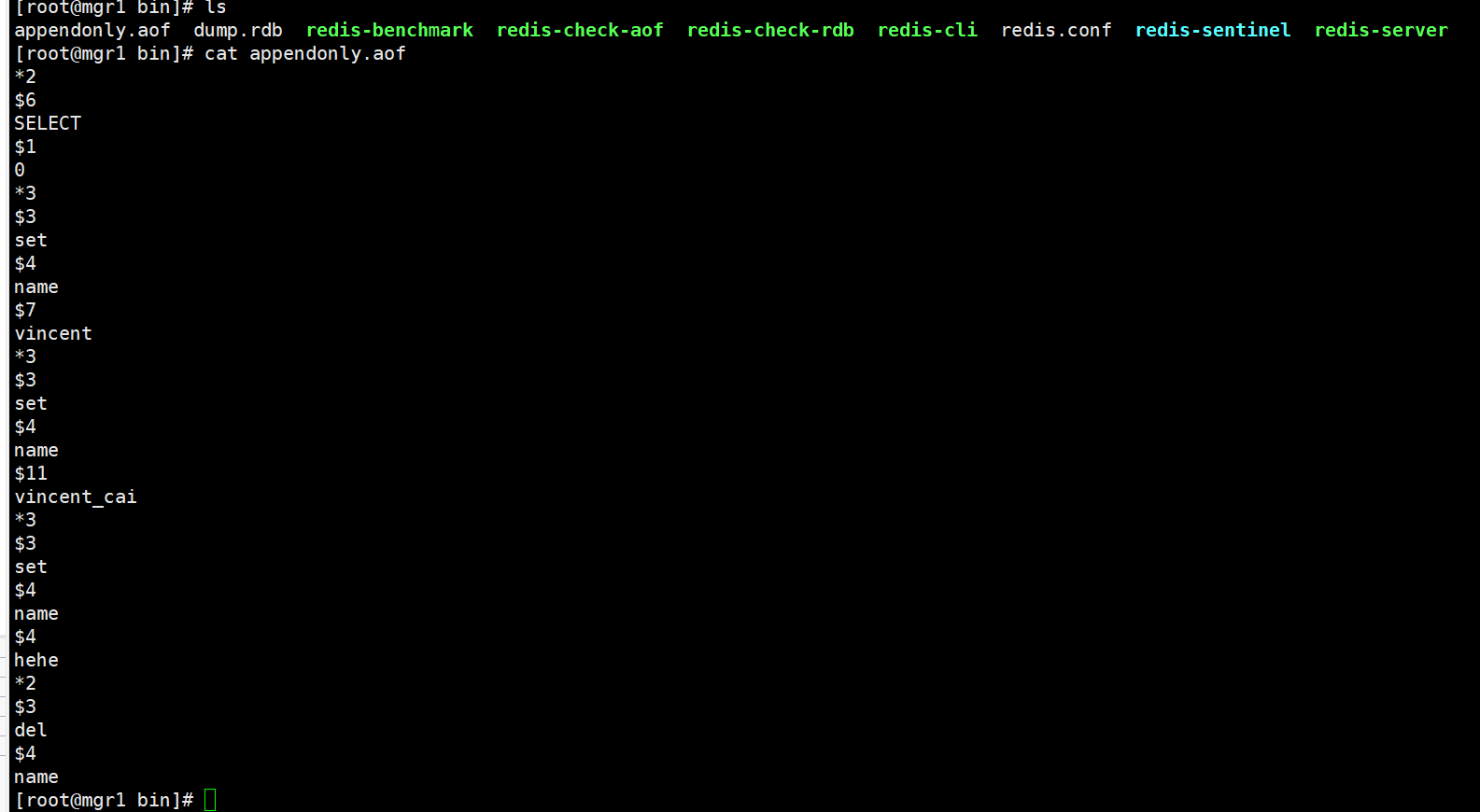
建议最好还是使用everysec 既能够保证数据的同步、效率也还可以。



Aof是以执行命令的形式实现同步

只保存会改变数据的命令

#### 测试



# Redis持久化机制

## 全量同步与增量同步的区别

全量同步：就是每天定时（避开高峰期）或者采用一个周期实现将数据拷贝到一个地方也就是Rdb存储。

增量同步：比如采用对行为的操作实现对数据的同步，也就是AOF。

全量与增量的比较：增量同步比全量同步更加消耗服务器的内存，但是能够更加的保证数据的同步。

## RDB与AOF实现持久化的区别

Redis提供了两种持久化的机制，分别为RDB、AOF实现，RDB采用定时（全量）持久化机制，但是服务器因为某种原因宕机后可能数据会丢失，AOF是基于数据日志操作实现的持久化，所以AOF采用增量同步的方案。

Redis已经帮助我默认开启了rdb存储。

RDB属于全量同步（定时同步）

优点：同步效率非常高

缺点：数据可能会丢失

AOF属于增量同步 有点偏实时

优点：同步效率比较低，最多只会丢失1秒数据

平衡点：既然要效率高，数据不丢失要使用aof的everysec

如何aof和rdb都启用则优先使用aof

# Redis内存淘汰策略

将Redis用作缓存时，如果内存空间用满，就会自动驱逐老的数据。

## Redis六种淘汰策略

noeviction：当内存使用达到阈值的时候，所有引起申请内存的命令会报错。(默认)

allkeys-lru：在主键空间中，优先移除最近未使用的key。(推荐)

volatile-lru：在设置了过期时间的键空间中，优先移除最近未使用的key。

allkeys-random：在主键空间中，随机移除某个key。

volatile-random：在设置了过期时间的键空间中，随机移除某个key。

volatile-ttl：在设置了过期时间的键空间中，具有更早过期时间的key优先移除。

noeviction：当内存使用达到阈值的时候，执行命令直接报错 (默认)

allkeys-lru：在所有的key中，优先移除最近未使用的key。(推荐)

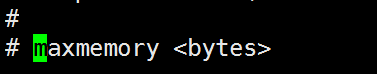
volatile-lru：在设置了过期时间的键空间中，优先移除最近未使用的key。

allkeys-random：在所有的key中，随机移除某个key。

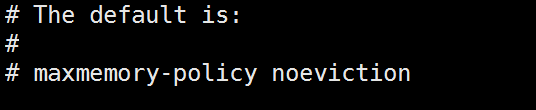
volatile-random：在设置了过期时间的键空间中，随机移除某个key。

volatile-ttl：在设置了过期时间的键空间中，具有更早过期时间的key优先移除。

## 如何配置Redis淘汰策略

在redis.conf文件中， 设置Redis 内存大小的限制，我们可以设置maxmemory <bytes>，当redis缓存中数据达到限定大小后，会选择配置的策略淘汰数据

比如：maxmemory 300mb。

通过配置 设置Redis的淘汰策略。比如：maxmemory-policy volatile-lru

# Redis中的自动过期机制

场景：创建订单，库存减1, ==》 支付

实现需求：处理订单过期自动取消，比如下单30分钟未支付自动更改订单状态

实现方案1：

1. 使用Redis Key自动过期出发事件通知
2. 使用定时任务30分钟后检查
3. 按照每分钟轮训检查

采用方案1 (这个模式有个致命的缺陷是，消息通知不能持久化，假如监听服务宕机期间，有key过期，那么这个失效通知就被忽略了。)

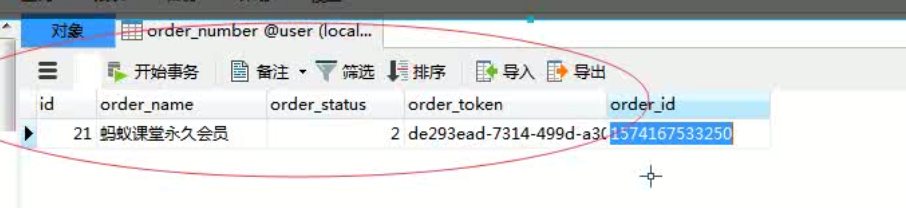
原理：

* 创建订单的时候绑定一个订单token存放在redis中(有效期只有30分钟)

Redis中存放 : Key = token value=订单id

即Key = dasdfsdf-7314-499d-a308-sdfsdf value=1567561465(订单ID)

* 对该Key绑定过期事件回调。在回调中检查order\_status是否为已支付，如果未支付，则将库存恢复



|  |
| --- |
| CREATE TABLE `order\_number` (  `id` int(11) NOT NULL AUTO\_INCREMENT,  `order\_name` varchar(255) DEFAULT NULL,  `order\_status` int(11) DEFAULT NULL,  `order\_token` varchar(255) DEFAULT NULL,  `order\_id` varchar(255) DEFAULT NULL,  PRIMARY KEY (`id`)  ) ENGINE=InnoDB AUTO\_INCREMENT=20 DEFAULT CHARSET=utf8; |

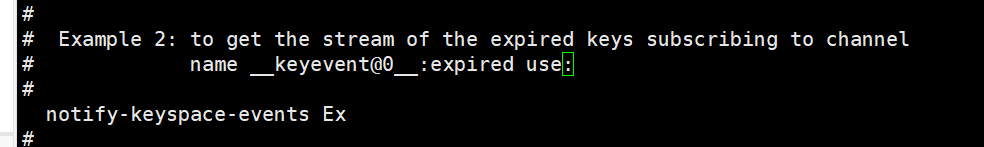
## 使用Redis Key自动过期机制

当我们的key失效时，可以执行我们的客户端回调监听的方法。

需要在Redis中配置：

notify-keyspace-events "Ex"

这样当Key过期时会发布该信息到Channel中，以供客户端订阅



## SpringBoot整合key失效监听

|  |
| --- |
| @Configuration **public class** RedisListenerConfig {  @Bean  RedisMessageListenerContainer container(RedisConnectionFactory connectionFactory) {  RedisMessageListenerContainer container = **new** RedisMessageListenerContainer();  container.setConnectionFactory(connectionFactory);  **return** container;  } } |

|  |
| --- |
| @Component **public class** RedisKeyExpirationListener **extends** KeyExpirationEventMessageListener {  **public** RedisKeyExpirationListener(RedisMessageListenerContainer listenerContainer) {  **super**(listenerContainer);  }   */\*\*  \* 待支付  \*/* **private static final** Integer ***ORDER\_STAYPAY*** = 0;  */\*\*  \* 失效  \*/* **private static final** Integer ***ORDER\_INVALID*** = 2;  @Autowired  **private** OrderMapper **orderMapper**;   */\*\*  \* Redis失效事件 key  \*  \** ***@param message*** *\** ***@param pattern*** *\*/* @Override  **public void** onMessage(Message message, **byte**[] pattern) {  String expiraKey = message.toString();  *// 根据key查询 value 如果还还是为待支付状态 将订单改为已经超时~~* OrderEntity orderNumber = **orderMapper**.getOrderNumber(expiraKey);  System.***out***.println(expiraKey);  **if** (orderNumber == **null**) {  **return**;  }  **if** (orderNumber.getOrderStatus().equals(***ORDER\_STAYPAY***)) {  *// 将订单状态改为已经失效* **orderMapper**.updateOrderStatus(expiraKey, ***ORDER\_INVALID***);  }  } } |

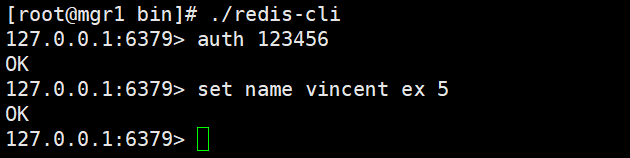
|  |
| --- |
| @RestController **public class** OrderController {  @Autowired  **private** OrderMapper **orderMapper**;  @Autowired  **private** RedisUtils **redisUtils**;   @RequestMapping(**"/saveOrder"**)  **public** String saveOrder() {  *// 1.生成token* String orderToken = UUID.*randomUUID*().toString();  String orderId = System.*currentTimeMillis*() + **""**;  *//2. 将该token存放到redis中* **redisUtils**.setString(orderToken, orderId, 5L);  OrderEntity orderEntity = **new** OrderEntity(**null**, **"蚂蚁课堂永久会员"**, orderId, orderToken);  **int** result = **orderMapper**.insertOrder(orderEntity);  **return** result > 0 ? **"success"** : **"fail"**;  } } |

|  |
| --- |
| @Data **public class** OrderEntity {  **private** Long **id**;  **private** String **orderName**;  */\*\*  \* 0 待支付 1 已经支付  \*/* **private** Integer **orderStatus**;   **private** String **orderToken**;  **private** String **orderId**;   **public** OrderEntity(Long id, String orderName, String orderId, String orderToken) {  **this**.**id** = id;  **this**.**orderName** = orderName;  **this**.**orderId** = orderId;  **this**.**orderToken** = orderToken;  } } |

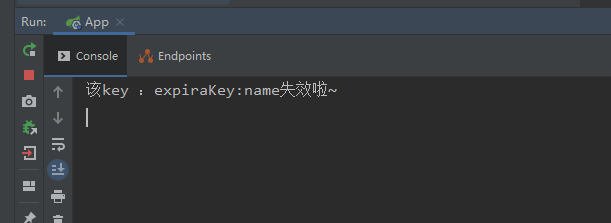
|  |
| --- |
| **public interface** OrderMapper {   @Insert(**"insert into order\_number values (null,#{orderName},0,#{orderToken},#{orderId})"**)  **int** insertOrder(OrderEntity OrderEntity);    @Select(**"SELECT ID AS ID ,order\_name AS ORDERNAME ,order\_status AS orderstatus,order\_token as ordertoken,order\_id as orderid FROM order\_number\n"** +  **"where order\_token=#{orderToken};"**)  OrderEntity getOrderNumber(String orderToken);   @Update(**"\n"** +  **"\n"** +  **"update order\_number set order\_status=#{orderStatus} where order\_token=#{orderToken};"**)  **int** updateOrderStatus(String orderToken, Integer orderStatus); } |

## 案例

设置一个失效的Key



客户端5秒后被回调

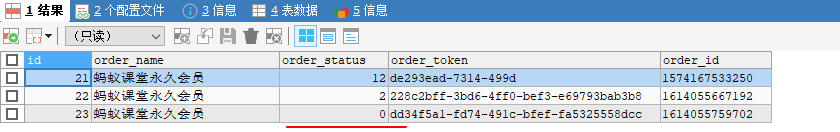


## 完整案例

发送请求

<http://localhost:8080/addOrder>

状态为未支付



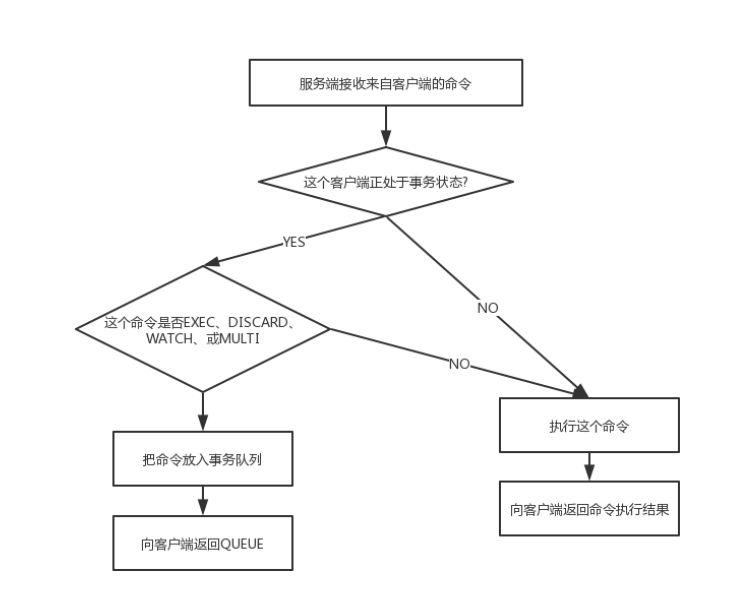
Key过期后token失效





# Redis事务操作

事务执行流程



命令：

**Multi 开启事务**

**EXEC 提交事务**

Watch 可以监听一个或者多个key，在提交事务之前是否有发生了变化 如果发生边了变化就不会提交事务，没有发生变化才可以提交事务。否则， **服务器将拒绝执行事务,并向客户端返回代表事务执行失败的空回复**

。 乐观锁， 通过版本号码实现

**Discard 取消提交事务**

案例：

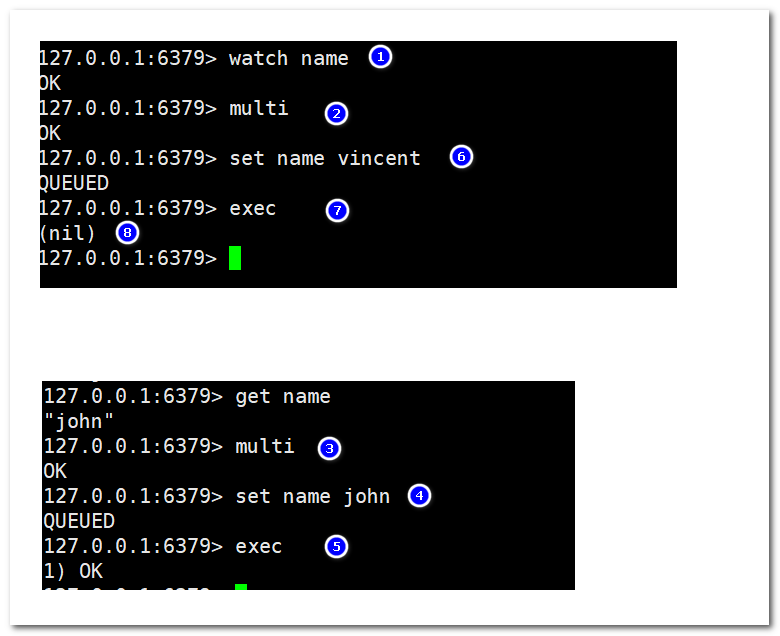
watch name

multi

set name xiaoxiao

exec

**Discard 取消提交事务**

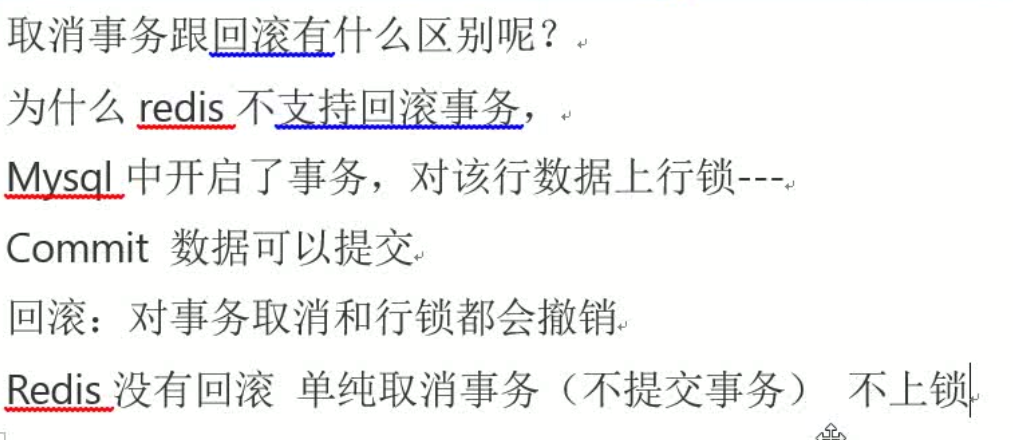


**注意：Redis官方是没有提供回滚方法， 只提供了取消事务。**

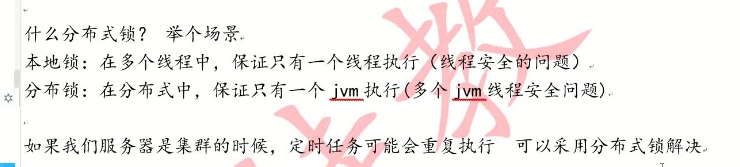
**Redis中本身就是单线程的能够保证线程安全问题。**

Redis和Mysql事务的区别:

Redis开启事务的时候，其他事务可以同时操作相同的Key，不能像Mysql一样支持行锁。



# Redis实现分布式锁



分布式锁实现方案：

* + - 1. 基于数据库方式实现
      2. 基于Zk方式实现，采用临时顺序结点+事件通知
      3. 基于Redis方式实现setnx

解决分布式锁核心思路：

* + - 1. 获取锁

多个不同的jvm同时创建一个标记，只要谁能够创建成功谁就能获得锁

* + - 1. 释放锁

释放该全局标记，其他的jvm重新进入到获取锁资源

* + - 1. 超时锁。避免死锁（1. 没有获取到锁的jvm怎么处理 2. 获取到锁的jvm怎么处理）
      * 等待获取锁的超时时间
      * 已经获取到锁 需要设置锁的有效期，比如5秒

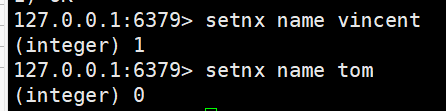
## Redis分布式锁实现思路

**分布式锁思路：**（这种思路仅限单体版本，集群有不同）

* + - 1. 获取锁

多个不同的jvm同时创建一个标记（使用setnx命令），因为Rediskey必须保证是唯一的，只要谁能够创建成功谁就能够获取锁

* + - * Set命令：如果key不存在则创建，如果key已经存在则修改原值
      * Setnx命令：如果key不存在则创建，如果不存在则不执行任何操作返回0



* + - 1. 释放锁

对我们的redis的key设置一个有效期(或者主动删除该key)可以灵活的自动释放该全局标记，其他的jvm重新进入到获取锁资源

* + - 1. 超时锁（1. 没有获取到锁的jvm怎么处理 2. 获取到锁的jvm怎么处理）
      * 等待获取锁的超时时间
      * 已经获取到锁 需要设置锁的有效期，比如5秒

**几个问题：**

* **如果业务流程没有执行完就因为超时释放了锁怎么办？**
  + 锁的超时时间首先要根据业务场景来预估
  + 可以自己延迟锁的时间(不可行)
  + 在提交事务的时候需要检查锁是否超时，如果已经超时了，则手动回滚；否则提交（可行）

**（这种思路仅限单体版本，**集群有不同**）**

Redis实现分布式锁基于SetNx命令，因为在Redis中key是保证是唯一的。所以当多个线程同时的创建setNx时，只要谁能够创建成功谁就能够获取到锁。

Set 命令 每次set时，可以修改原来旧值；

SetNx命令 每次SetNx检查该 key是否已经存在，如果已经存在的话不会执行任何操作。返回为0 如果已经不存在的话直接新增该key。

1：新增key成功 0 失败

获取锁的时候：当多个线程同时创建SetNx k，只要谁能够创建成功谁就能够获取到锁。

释放锁：可以对该key设置一个有效期可以避免死锁的现象。

Redis官网有一个redis解决分布式锁的框架： Redission

## Zookeeper实现分布式锁思路

Zookeeper实现分布式锁核心采用临时顺序节点+事件通知，因为zk节点路径是保证全局唯一的，当多个线程同时创建该临时节点，只要谁能够创建成功谁就能够获取到锁。

* 临时结点可以设置有效期

获取锁：当多个线程同时创建该临时节点，只要谁能够创建成功谁就能够获取到锁。

释放锁：关闭当前Session连接；或者超过了Session有效期；自动的删除当前的zk节点路径，其他线程重新进入到获取锁阶段。

## 分布式锁的应用场景有那些

1. 分布式任务调度平台保证任务的幂等性。
2. 分布式全局id的生成

## Redis分布式锁核心代码

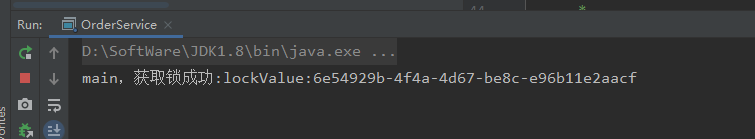
|  |
| --- |
| **public class** RedisUtil {  *//protected static Logger logger = Logger.getLogger(RedisUtil.class);* **private static** String *IP* = **"192.168.212.148"**;   *//Redis的端口号* **private static int** *PORT* = 6379;   *//可用连接实例的最大数目，默认值为8；  //如果赋值为-1，则表示不限制；如果pool已经分配了maxActive个jedis实例，则此时pool的状态为exhausted(耗尽)。* **private static int** *MAX\_ACTIVE* = 100;   *//控制一个pool最多有多少个状态为idle(空闲的)的jedis实例，默认值也是8。* **private static int** *MAX\_IDLE* = 20;   *//等待可用连接的最大时间，单位毫秒，默认值为-1，表示永不超时。如果超过等待时间，则直接抛出JedisConnectionException；* **private static int** *MAX\_WAIT* = 3000;   **private static int** *TIMEOUT* = 3000;   *//在borrow一个jedis实例时，是否提前进行validate操作；如果为true，则得到的jedis实例均是可用的；* **private static boolean** *TEST\_ON\_BORROW* = **true**;   *//在return给pool时，是否提前进行validate操作；* **private static boolean** *TEST\_ON\_RETURN* = **true**;   **private static** JedisPool *jedisPool* = **null**;   */\*\*  \* redis过期时间,以秒为单位  \*/* **public final static int *EXRP\_HOUR*** = 60 \* 60; *//一小时* **public final static int *EXRP\_DAY*** = 60 \* 60 \* 24; *//一天* **public final static int *EXRP\_MONTH*** = 60 \* 60 \* 24 \* 30; *//一个月   /\*\*  \* 初始化Redis连接池  \*/* **private static void** initialPool() {  **try** {  JedisPoolConfig config = **new** JedisPoolConfig();  config.setMaxTotal(*MAX\_ACTIVE*);  config.setMaxIdle(*MAX\_IDLE*);  config.setMaxWaitMillis(*MAX\_WAIT*);  config.setTestOnBorrow(*TEST\_ON\_BORROW*);   *jedisPool* = **new** JedisPool(config, *IP*, *PORT*, *TIMEOUT*, **"123456"**);  } **catch** (Exception e) {  *//logger.error("First create JedisPool error : "+e);* e.getMessage();  }  }    */\*\*  \* 在多线程环境同步初始化  \*/* **private static synchronized void** poolInit() {  **if** (*jedisPool* == **null**) {  *initialPool*();  }  }    */\*\*  \* 同步获取Jedis实例  \*  \** ***@return*** *Jedis  \*/* **public synchronized static** Jedis getJedis() {  **if** (*jedisPool* == **null**) {  *poolInit*();  }  Jedis jedis = **null**;  **try** {  **if** (*jedisPool* != **null**) {  jedis = *jedisPool*.getResource();  }  } **catch** (Exception e) {  e.getMessage();  *// logger.error("Get jedis error : "+e);* }  **return** jedis;  }    */\*\*  \* 释放jedis资源  \*  \** ***@param jedis*** *\*/* **public static void** returnResource(**final** Jedis jedis) {  **if** (jedis != **null** && *jedisPool* != **null**) {  *jedisPool*.returnResource(jedis);  }  }   **public static** Long sadd(String key, String... members) {  Jedis jedis = **null**;  Long res = **null**;  **try** {  jedis = *getJedis*();  res = jedis.sadd(key, members);  } **catch** (Exception e) {  *//logger.error("sadd error : "+e);* e.getMessage();  }  **return** res;  } } |

|  |
| --- |
| **public class** MayiktRedisLock {   **private static final int *setnxSuccss*** = 1;   */\*\*  \* 获取锁  \*  \** ***@param lockKey*** *定义锁的key  \** ***@param notLockTimeOut*** *没有获取锁的超时时间  \** ***@param lockTimeOut*** *使用锁的超时时间  \** ***@return*** *\*/* **public** String getLock(String lockKey, **int** notLockTimeOut, **int** lockTimeOut) {  *// 获取Redis连接* Jedis jedis = RedisUtil.*getJedis*();  *// 定义没有获取锁的超时时间* Long endTimeOut = System.*currentTimeMillis*() + notLockTimeOut;  **while** (System.*currentTimeMillis*() < endTimeOut) {  String lockValue = UUID.*randomUUID*().toString();  *// 如果在多线程情况下谁能够setnx 成功返回0 谁就获取到锁* **if** (jedis.setnx(lockKey, lockValue) == ***setnxSuccss***) {  jedis.expire(lockKey, lockTimeOut / 1000);  **return** lockValue;  }  *// 否则情况下 在超时时间内继续循环* }  **try** {  **if** (jedis != **null**) {  jedis.close();  }  } **catch** (Exception e) {  e.printStackTrace();  }  **return null**;  }   */\*\*  \* 释放锁 其实就是将该key删除  \*  \** ***@return*** *\*/* **public** Boolean unLock(String lockKey, String lockValue) {  Jedis jedis = RedisUtil.*getJedis*();  *// 确定是对应的锁 ，才删除* **if** (lockValue.equals(jedis.get(lockKey))) {  **return** jedis.del(lockKey) > 0 ? **true** : **false**;  }  **return false**;  } } |

|  |
| --- |
| **public class** OrderService {   **private** MayiktRedisLock **mayiktRedisLock** = **new** MayiktRedisLock();  **private** String **lockKey** = **"mayikt\_lock"**;   **public void** service() {  *// 1.获取锁* String lockValue = **mayiktRedisLock**.getLock(**lockKey**, 5000, 5000);  **if** (StringUtils.*isEmpty*(lockValue)) {  System.***out***.println(Thread.*currentThread*().getName() + **",获取锁失败!"**);  **return**;  }  *// 2.获取锁成功执行业务逻辑* System.***out***.println(Thread.*currentThread*().getName() + **",获取成功，lockValue："** + lockValue); *// // 3.释放lock锁* **mayiktRedisLock**.unLock(**lockKey**, lockValue);  } } |

## 测试

* 设置持有锁时间5S，获取锁最多等待5S。 启动程序A后，会获取锁成功。过2秒后，再次启动程序A后，会等待3秒，然后又可以获取锁。



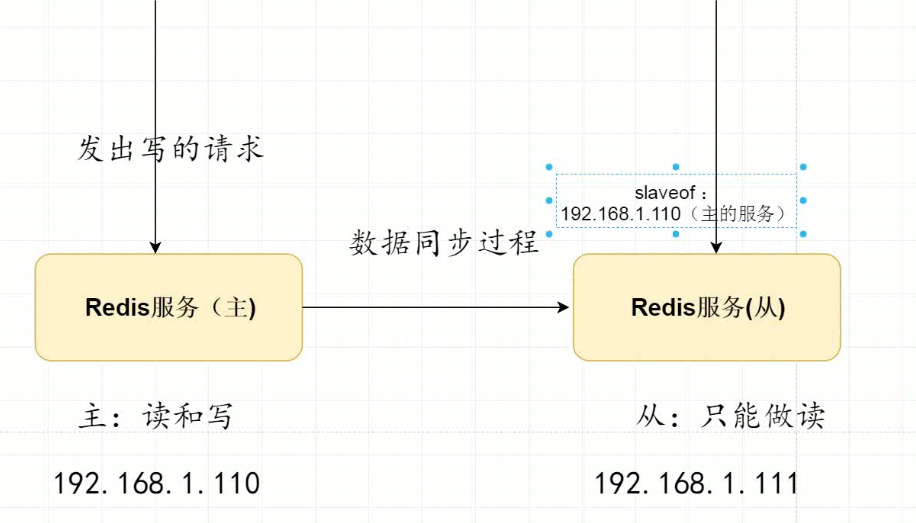
* 如果没有给锁设置超时时间，并且忘记了释放锁，则会进入死锁状态。！！！

# Redis集群高可用环境

## Redis主从复制

基本概念：

单个Redis如果因为某种原因宕机的话，可能会导致Redis服务不可用，可以使用主从复制实现一主多从，主节点负责写的操作，从节点负责读的操作，主节点会定期将数据同步到从节点中，保证数据一致性的问题。



相关配置从服务器的Redis.conf

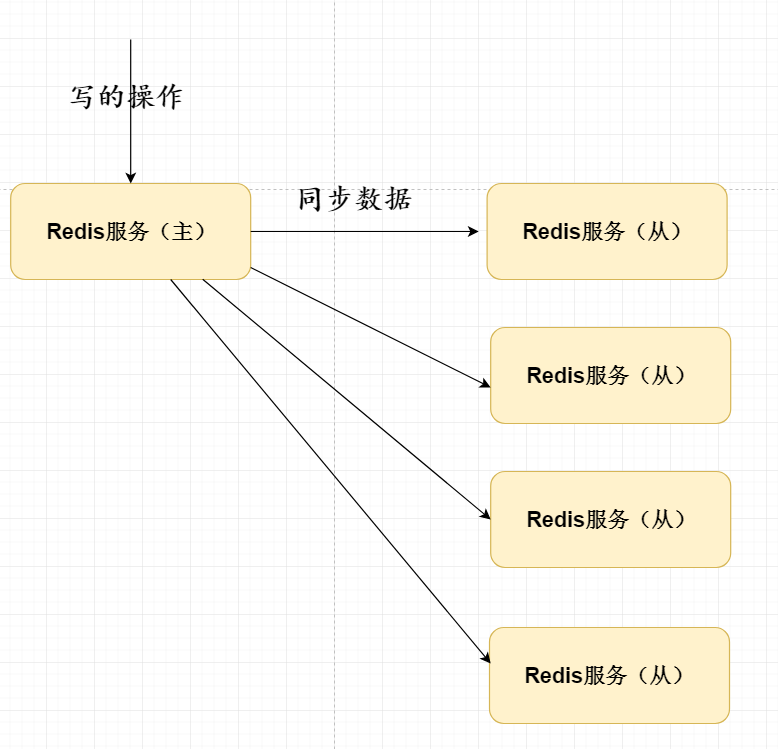
# replicaof <masterip> <masterport> （官方因为slave是敏感词，改为了replicaof，但是slaveof依然兼容）

slaveof 192.168.212.160 6379

masterauth 123456 （主服务器的密码，如果有才要配置）

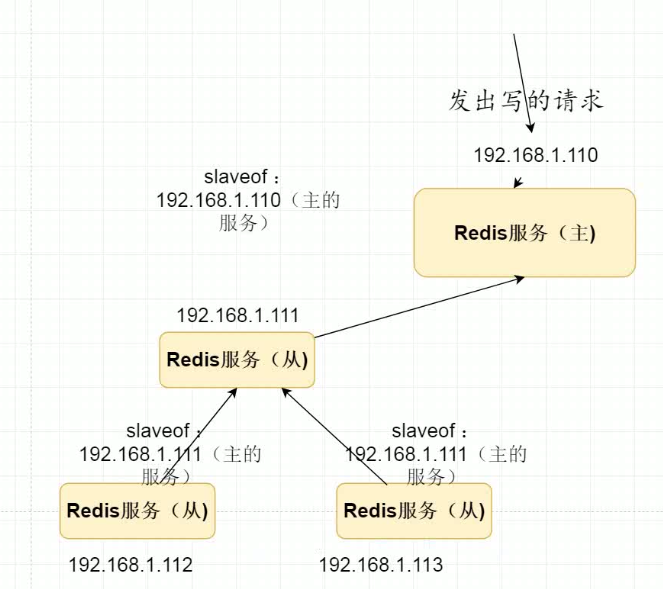
测试：在主服务器或者从服务器执行

info replication (获取从服务器信息)



该主从同步方式存在 如果从节点非常多的话，会导致对主节点同步多个从节点压力非常大

可以采用树状类型解决该问题



### 主从复制数据同步的过程

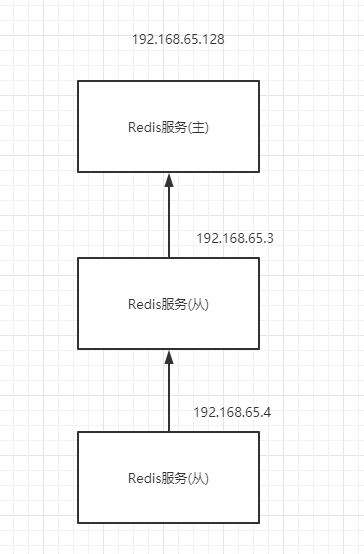
1. 需要在从服务器配置在slaveof <masterip> <masterport> ；以及配置密码
2. Redis从节点向主节点建立socket长连接
3. Redis采用全量或者增量的形式将数据同步给从节点

从Redis2.8版本以后 过程采用增量和全量同步

全量复制：一般用于在初次的复制场景（从节点与主节点一次建立）(主服务器执行BGSAVE生成二进制dump.rdb文件，并发送给从服务器；并将缓冲区所有写命令也发送给从服务器，从而达到初始一致)

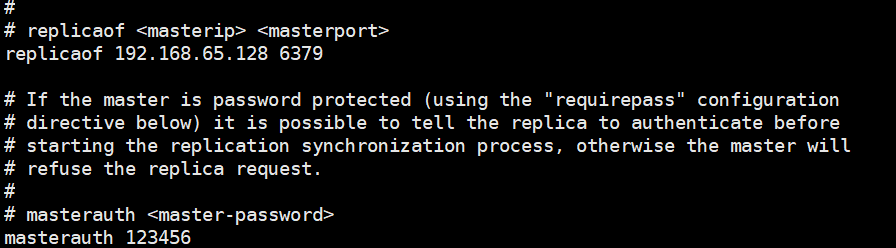
增量复制：网络出现问题，从节点再次连接主节点时，主节点补发缺少的数据，每次数据增量同步

### 案例：搭建一个主从集群，1主2从



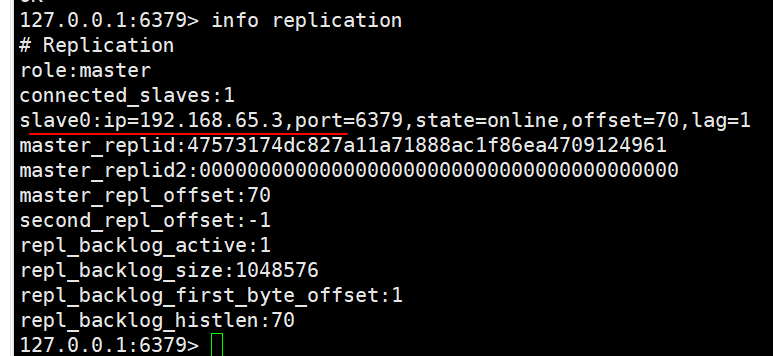
#### 搭建主服务器192.168.65.128

#### 搭建从服务器 192.168.65.3

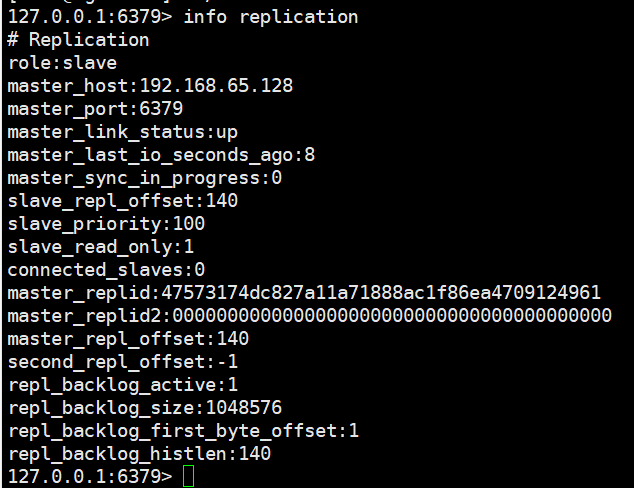


#### 测试主从服务器

* 主服务器

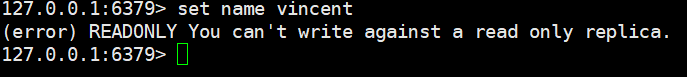


* 从服务器

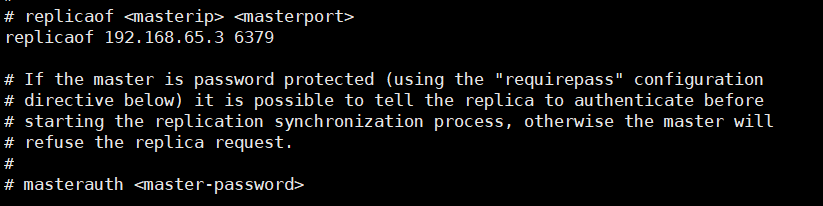


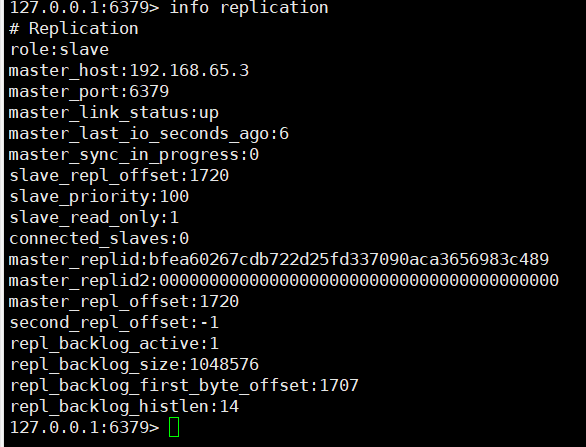
#### 测试从服务器能否执行写命令

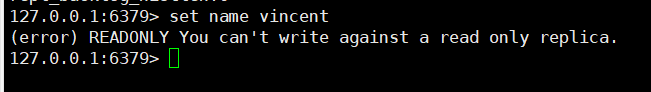
从服务器不能执行写操作



#### 搭建从服务器192.168.65.4



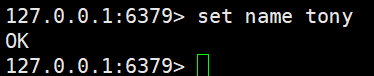




#### 主服务器发128发一条消息到两个从服务器

从服务器能够成功同步消息

主128：



从65.3：



从65.4：



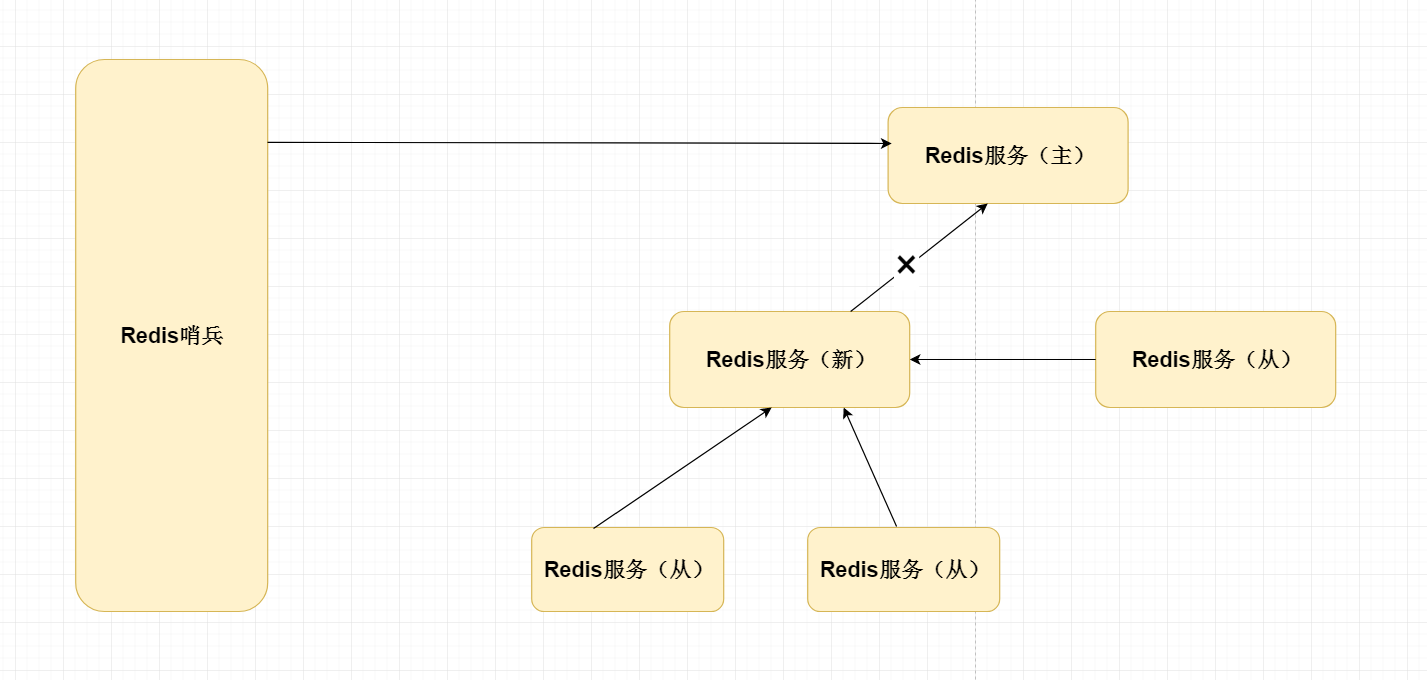
### 主从复制存在那些缺陷

如果主节点存在了问题，整个Redis环境是不可以实现写的操作，需要人工更改配置变为主操作

如何解决该问题：使用哨兵机制可以帮助解决Redis集群主从选举策略。

## Redis哨兵机制

Redis的哨兵机制就是解决我们以上主从复制存在缺陷（选举问题），解决问题保证我们的Redis高可用，实现自动化故障发现与故障转移。



### 哨兵机制原理

1. 哨兵机制每个10s时间只需要配置监听我们的主节点就可以获取当前整个Redis集群的环境列表，采用info 命令形式。
2. 哨兵不建议是单机的，最好每个Redis节点都需要配置哨兵监听。
3. 哨兵集群原理是如何：多个哨兵都会检测同一个主的master节点，订阅到相同的通道，有新的哨兵加入都会向通道中发送自己服务的信息，该通道的订阅者可以发现新哨兵的加入，随后相互建立长连接。
4. Master的故障发现 单个哨兵会向主的master节点发送ping的命令，如果master节点没有及时的响应，哨兵会认为该master节点为“主观不可用状态（需要让其他哨兵确认）”会发送给其他都哨兵确认该Master节点是否不可用，当前确认的哨兵节点数>=quorum(可配置)，会实现重新选举。

### 相关核心配置

* 将sentinel.conf拷贝到相关目录，后面要使用该文件来启动哨兵

cp /usr/redis-5.0.6/sentinel.conf /usr/redis/bin

cd /usr/redis/bin

* 修改配置，让哨兵监听主服务

vi sentinel.conf

sentinel monitor mymaster 192.168.65.128 6379 2 (ip 端口号 需要几个哨兵确认)

sentinel auth-pass mymaster 123456

* 可配置的监听相关的配置

sentinel down-after-milliseconds mymaster 3000#sentinel心跳检测主3秒内无响应，视为挂掉，开始切换其他从为主

sentinel parallel-syncs mymaster 1#每次最多可以有1个从同步主。一个从同步结束，另一个从开始同步。

sentinel failover-timeout mymaster 18000#主从切换超时时间

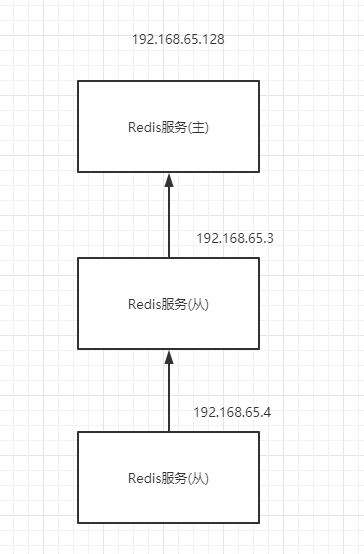
* 启动哨兵

启动哨兵./redis-sentinel ./sentinel.conf

通过实践发现的问题：哨兵只帮我们做了选举，以及让下线的主服务重连后变为重服务。但是没有帮我们处理主从复制的问题。即主服务65.128重连后，无法收到新的主服务65.3的缓存更新。

### 案例：

#### 服务关系如下，在每台机器上创建哨兵



#### 配置哨兵

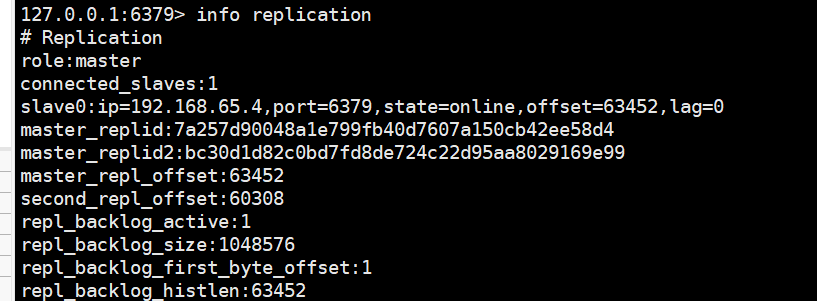
|  |
| --- |
| vi sentinel.conf  sentinel monitor mymaster 192.168.65.128 6379 2 (ip 端口号 需要几个哨兵确认)  sentinel auth-pass mymaster 123456 |

#### 测试自动选举与恢复上线

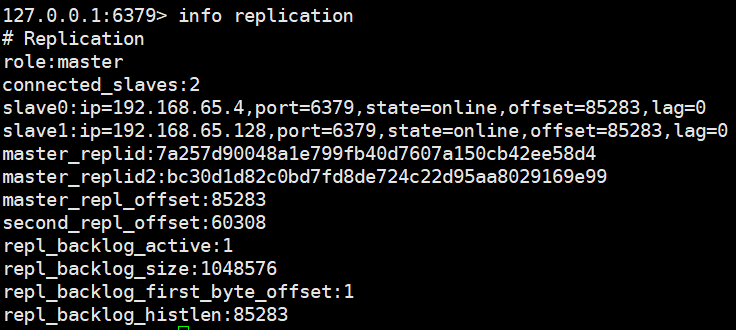
* 关掉主服务器的redis服务 192.168.65.128



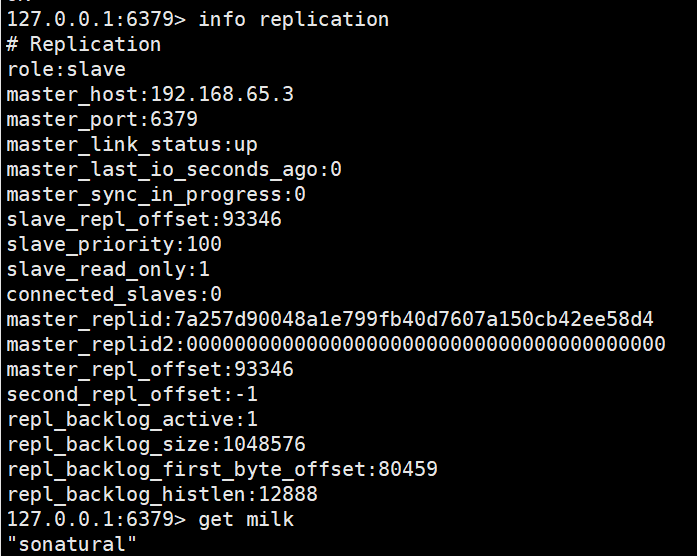
* 发现从服务器192.168.65.3升级为了主服务器



* 测试192.168.65.128重新上线后是否为变为从服务器
  + 可以看到192.168.65.3下有2台从服务器



* + 可以看到192.168.65.128变为从服务器，并且可以读到最新的缓存数据



#### 查看相关配置文件

* Sentinel.conf

|  |
| --- |
| 192.168.65.128 的配置文件会被哨兵修改，指向新的主节点 |

|  |
| --- |
| 192.168.65.4 的配置文件也是如此 |

* Reids.conf

|  |
| --- |
| 192.168.65.128 |

|  |
| --- |
| 192.168.65.4 |

# Redis安全控制

## 缓存穿透

产生的背景：类似DDOS攻击，通过大量不能命中缓存的KEY 访问服务，导致数据库压力过大

缓存穿透是指使用不存在的key进行大量的高并发查询，导致缓存无法命中，每次请求都要都要穿透到后端数据库查询，使得数据库的压力非常大，甚至导致数据库服务压死；

解决方案：

1. 接口层实现api限流、用户授权、id检查等 黑名单和白名单；
2. 从缓存和数据库都取不到数据的话，将该空值Key放入缓存中，设置短时间(30s)有效期避免使用同一个id对数据库攻击压力大；
3. 使用布隆过滤器

## 缓存击穿

产生背景：热点Key突然失效，因为访问该Key的请求过多，此时多个请求会同时查询数据库。并把数据库内容放入到缓存。

在高并发的情况下，当一个缓存key过期时，因为访问该key请求较大，多个请求同时发现缓存过期，因此对多个请求同时数据库查询、同时向Redis写入缓存数据，这样会导致数据库的压力非常大；

解决方案：

1. 使用分布式锁（集群环境）

保证在分布式情况下，使用分布式锁保证对于每个key同时只允许只有一个线程查询到后端服务，其他没有获取到锁的权限，只需要等待即可；这种高并发压力直接转移到分布式锁上，对分布式锁的压力非常大。

1. 使用本地锁(单机环境)

使用本地锁与分布式锁机制一样，只不过分布式锁适应于服务集群、本地锁仅限于单个服务使用。

1. 软过过期

设置热点数据永不过期或者异步延长过期时间；

## 缓存雪崩

产生背景：

缓存雪崩指缓存服务器重启或者大量的缓存集中在某个时间段失效，突然给数据库产生了巨大的压力，甚至击垮数据库的情况。

解决思路：

* 对不用的数据使用不同的失效时间，加上随机数

## 其他方面也要注意：

* 做好服务降级，服务熔断
* 集群化部署，分摊访问压力
* 做好监控

穿透：当查询一个key 不存在的情况

雪崩：多个key同时失效情况

击穿：单个key失效的情况下

# Redis Cluster集群

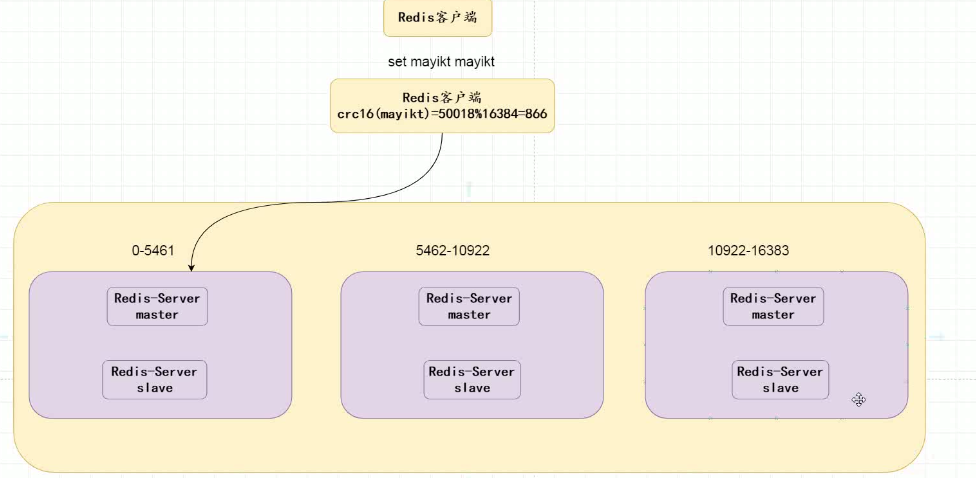
## 传统Redis集群存在那些问题

Redis哨兵集群模式，每个节点都保存全量同步数据，冗余的数据比较多；而在Redis Cluster模式中集群中采用分片集群模式，可以减少冗余数据，缺点就是构建该集群模式成本非常高，

## 传统RedisCluster集群的原理

**最大特点：去中心化**

Redis3.0开始官方推出了集群模式 RedisCluster，原理采用hash槽的概念，预先分配16384个卡槽，并且将该卡槽分配给具体服务的节点；通过key进行crc16（key）%16384 获取余数，余数就是对应的卡槽的位置，一个卡槽可以存放多个不同的key，从而将读或者写转发到该卡槽的服务的节点。 最大的有点：动态扩容、缩容。



## RedisCluster集群模式环境搭建

|  |
| --- |
| mkdir rediscluster  cd rediscluster/  mkdir redis7000  mkdir redis7001  mkdir redis7002  mkdir redis7003  mkdir redis7004  mkdir redis7005  每个配置文件内容  daemonize yes #后台启动  protected-mode no ; ## 允许外部访问  port 7005 #修改端口号，从7000到7005  cluster-enabled yes #开启cluster，去掉注释  cluster-config-file 7000nodes.conf #自动生成  cluster-node-timeout 15000 #节点通信时间  logfile  /usr/rediscluster/redis7005/redis.log  启动我们的redis  /usr/redis/bin/redis-server /usr/rediscluster/redis7000/redis.conf  /usr/redis/bin/redis-server /usr/rediscluster/redis7001/redis.conf  /usr/redis/bin/redis-server /usr/rediscluster/redis7002/redis.conf  /usr/redis/bin/redis-server /usr/rediscluster/redis7003/redis.conf  /usr/redis/bin/redis-server /usr/rediscluster/redis7004/redis.conf  /usr/redis/bin/redis-server /usr/rediscluster/redis7005/redis.conf  连接一个redis  /usr/redis/bin/redis-cli -h 192.168.65.128 -p 7000    (error) CLUSTERDOWN Hash slot not served 说明没有分配hash槽  /usr/redis/bin/redis-cli --cluster create 192.168.65.128:7000 192.168.65.128:7001 192.168.65.128:7002 192.168.65.128:7003 192.168.65.128:7004 192.168.65.128:7005 --cluster-replicas 1  (建议最好使用服务器的ip地址搭建)  分配我们hash操作      重新运行  /usr/redis/bin/redis-cli -h 192.168.65.128 -p 7000  发现默认没有给我们进行重定向    添加-c参数，指明自动重定向  修改为Redis的集群方式连接/usr/redis/bin/redis-cli -h 192.168.65.128 -p 7000 -c (重定向)    查看7002的从结点7004，查看是否能从该从结点中读取key  发现不能，会重定向到Master结点7002    让我们看看7002宕机的情况下，7004能不能升级为主节点进行工作。  结论：能够有效的进行选举并进行失效转移        Cluster相关指令如下  /usr/redis/bin/redis-cli --cluster help  相关命令可以查看官网：  https://www.cnblogs.com/zhoujinyi/p/11606935.html  https://redis.io/topics/cluster-tutorial |
|  |
|  |

## RedisCluster集群模式扩容缩容节点

### 模拟扩容

该案例演示加入2台服务器，其中7006作为主服务器，7007作为7006的从服务器。

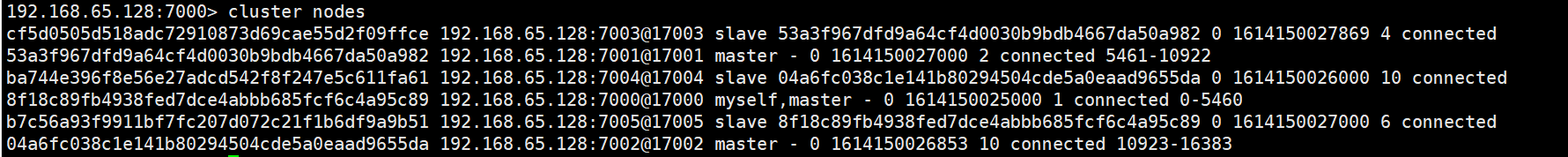
* 为了模拟扩容，启动Redis服务 。 7006(主)，7007(从)

/usr/redis/bin/redis-server /usr/rediscluster/redis7006/redis.conf

/usr/redis/bin/redis-server /usr/rediscluster/redis7007/redis.conf

* 查看当前集群状态

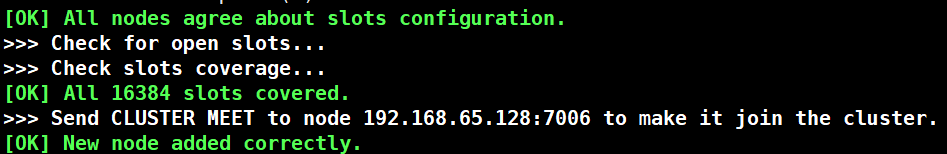
/usr/redis/bin/redis-cli -h 192.168.65.128 -p 7000



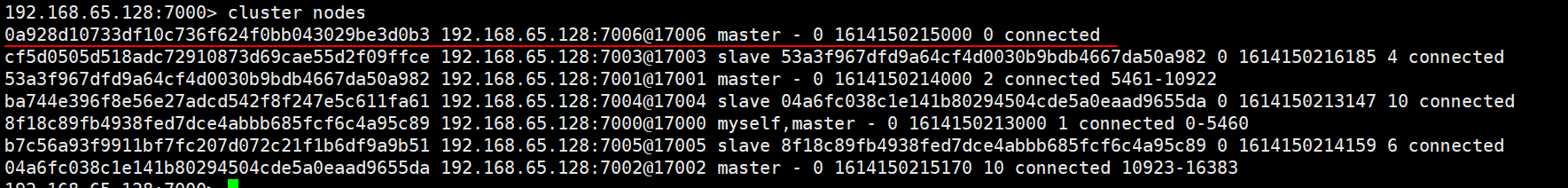
此时发现7006 7007没有加入到集群，需要通过客户端指定

* 将结点7006添加到集群

/usr/redis/bin/redis-cli --cluster add-node 192.168.65.128:7006 192.168.65.128:7000



* 将结点7007添加到集群，并设置7006为其Master结点
  + 查看7006结点的ID



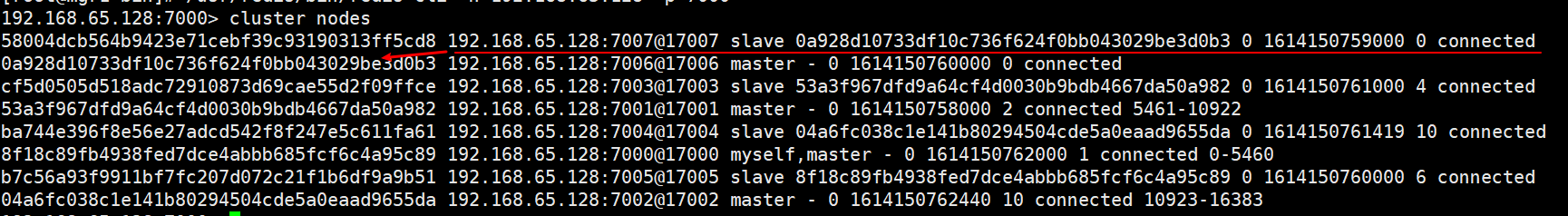
* + 将结点7007添加到集群，并设置7006为其Master结点

/usr/redis/bin/redis-cli --cluster add-node 192.168.65.128:7007 192.168.65.128:7000 --cluster-slave --cluster-master-id 0a928d10733df10c736f624f0bb043029be3d0b3

|  |
| --- |
| <https://redis.io/topics/cluster-tutorial> |

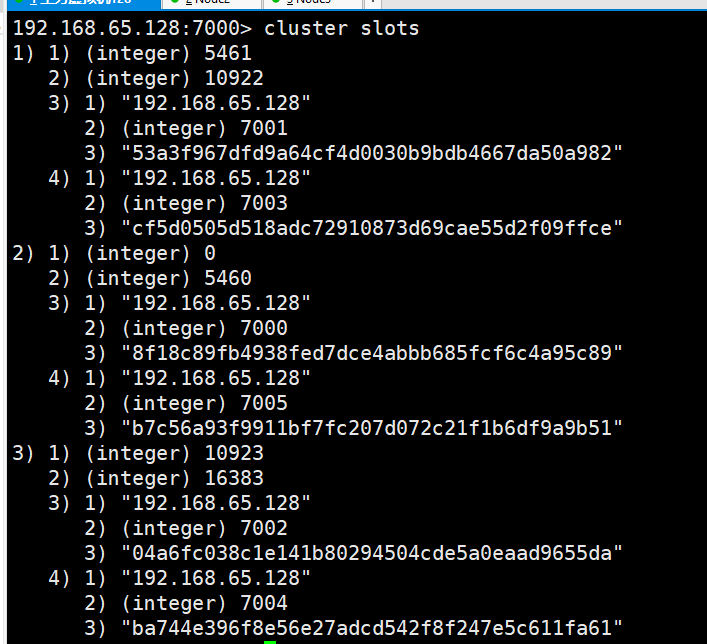
* + 查看集群状态

可以看见7007已经添加到集群中，并设置好了MASTER



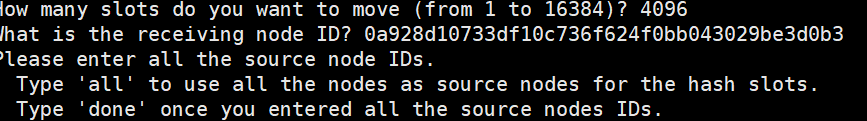
* 分配Redis槽位，进行扩容： 新增的7006 是没有任何槽位

cluster slots #查看槽

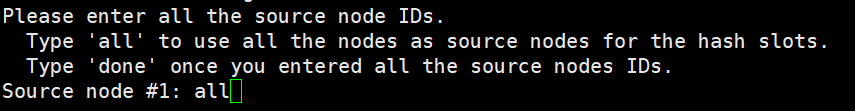


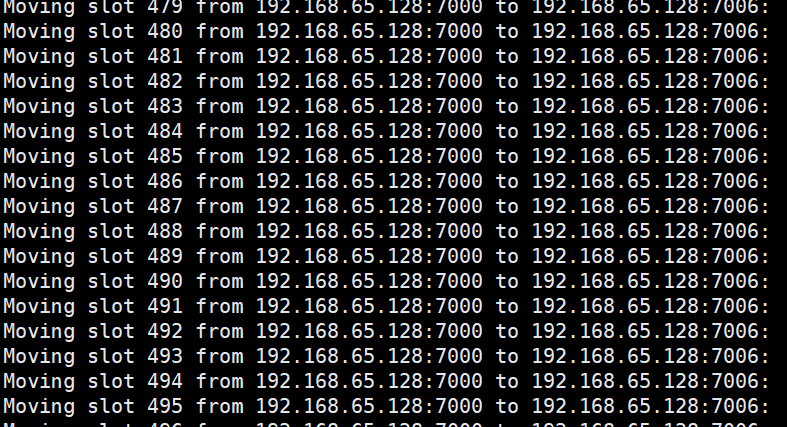
/usr/redis/bin/redis-cli --cluster reshard 192.168.65.128:7000 #连接到某一个客户端，让集群进行重新分配卡槽

* + - 指定需要移动多少个槽位 （这个案例中有4个结点，一共有16384个槽，每个结点应该有4096个槽）
    - 指定目的地(接收槽)的node ID (7006)

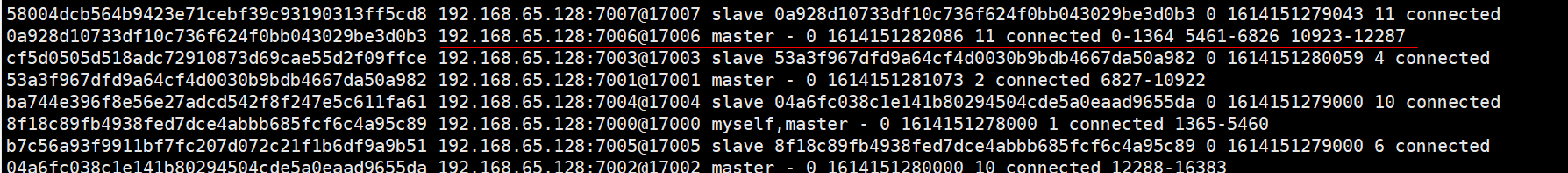


* + - 指定从哪些结点迁移相应的槽到目的地(7006)





* 查看结果，可以发现7006结点已经接收了来自其他结点分配的槽

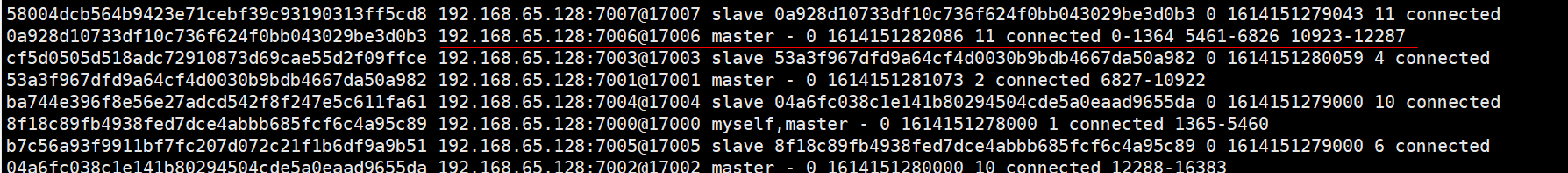


### 模拟缩容

Redis槽位缩容

该案例模拟7006结点下线。并将7006的槽迁移到7000

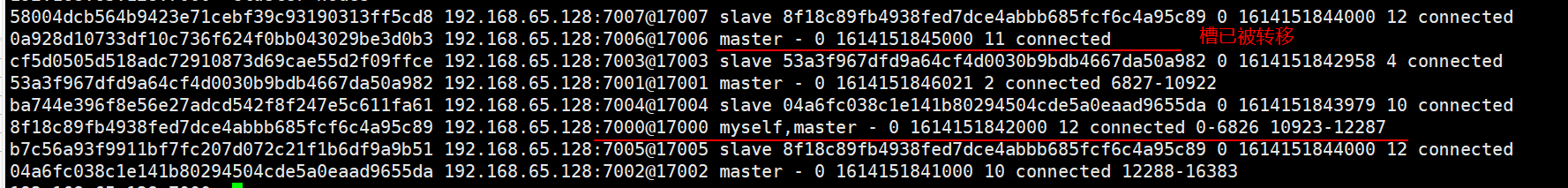
查看集群状态



/usr/redis/bin/redis-cli --cluster reshard 192.168.65.128:7000 --cluster-from 0a928d10733df10c736f624f0bb043029be3d0b3 --cluster-to 8f18c89fb4938fed7dce4abbb685fcf6c4a95c89 --cluster-slots 4096

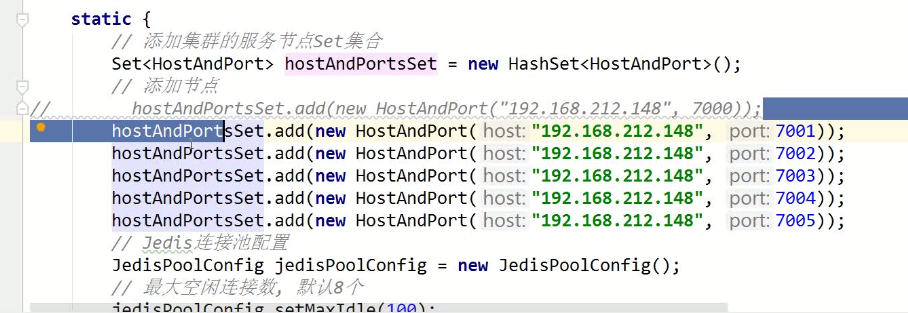
|  |
| --- |
| https://redis.io/topics/cluster-tutorial |

然后通过cluster nodes查看7000是否获得了转移的卡槽



### 通过Jedis连接Redis集群

连接一个或多个结点都可以，多个结点可以进行容错。原理是客户端发送命令后，进行crc16哈希算法计算Key对应的槽，并重定向到响应服务器执行，最后将结果返回。



# 基于布隆过滤器解决缓存穿透问题

## 缓存穿透

产生的背景：

缓存穿透是指使用不存在的key进行大量的高并发查询，导致缓存无法命中，每次请求都要都要穿透到后端数据库查询，使得数据库的压力非常大，甚至导致数据库服务压死；

解决方案：

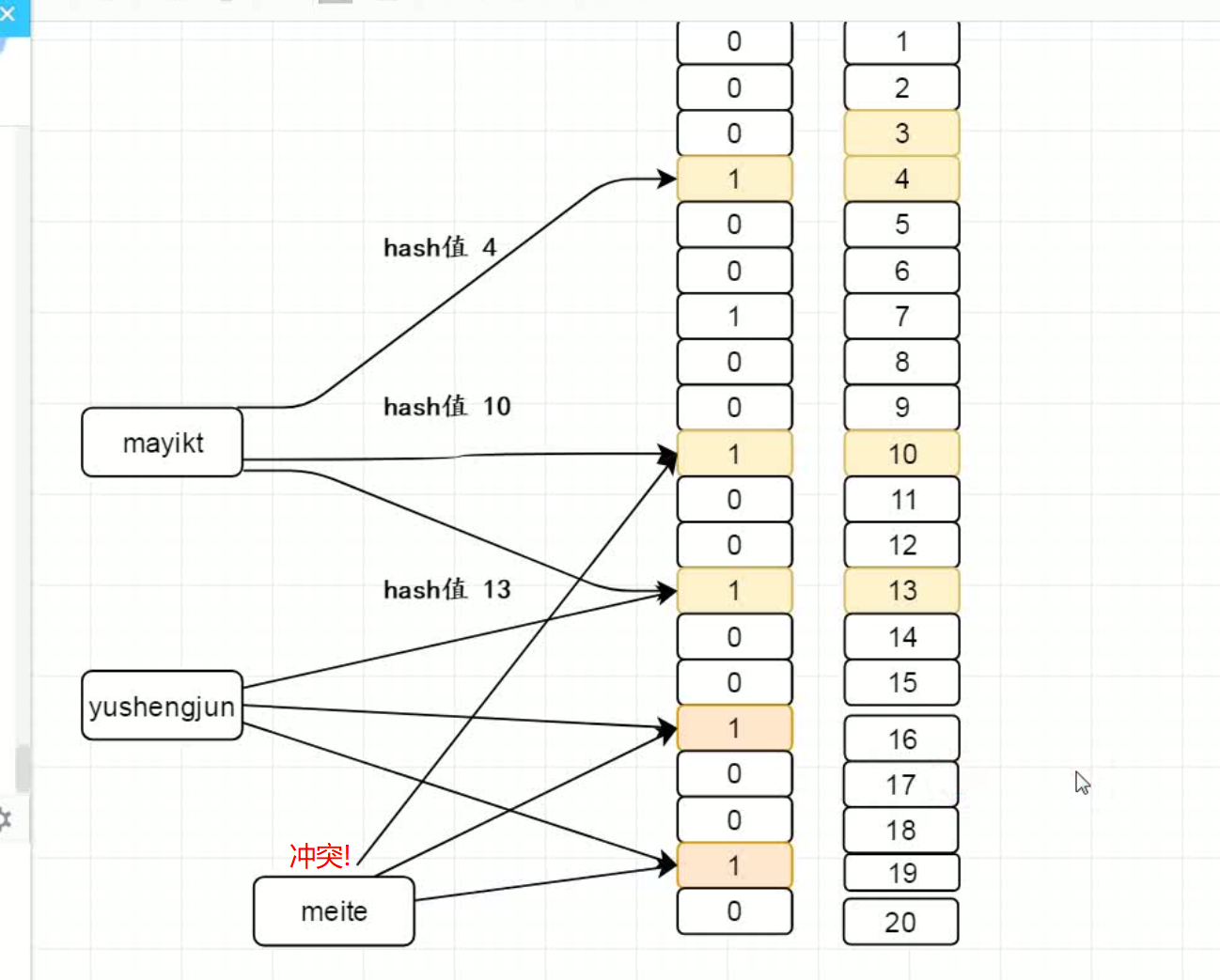
1. 接口层实现api限流、用户授权、id检查等；
2. 从缓存和数据库都取不到数据的话，一样将数据库空值放入缓存中，设置30s有效期避免使用同一个id对数据库攻击压力大；

## 布隆过滤器基本介绍

布隆过滤器适用于判断某个元素是否在集合中存在，不一定百分百正确，存在一定几率误判( 某个元素不在集合中，但布隆过滤器会判断它在集合中 )。 Bloom Filter基本实现原理采用二进制位数组与联合函数(多个哈希函数)一起实现；

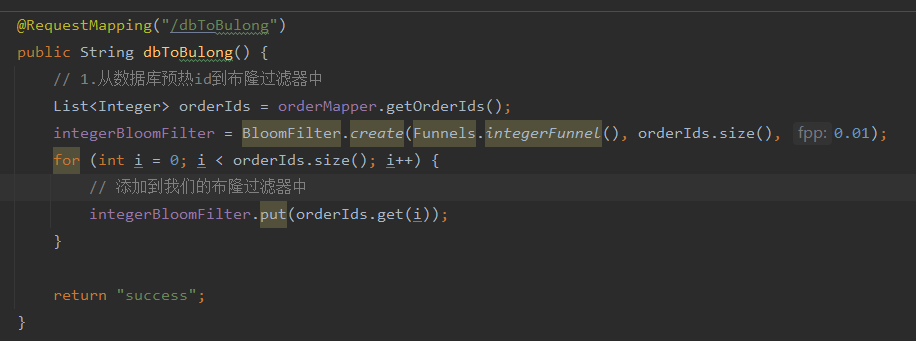
误判率基本可控制在可接受的范围。如果要提高精度，则底层会增加位数组长度。

布隆过滤器为什么会产生冲突，多个哈希函数会根据key计算hash值，并将对应位置的值设为1 . 这可能正好与布隆过滤器中已存放的元素产生冲突。

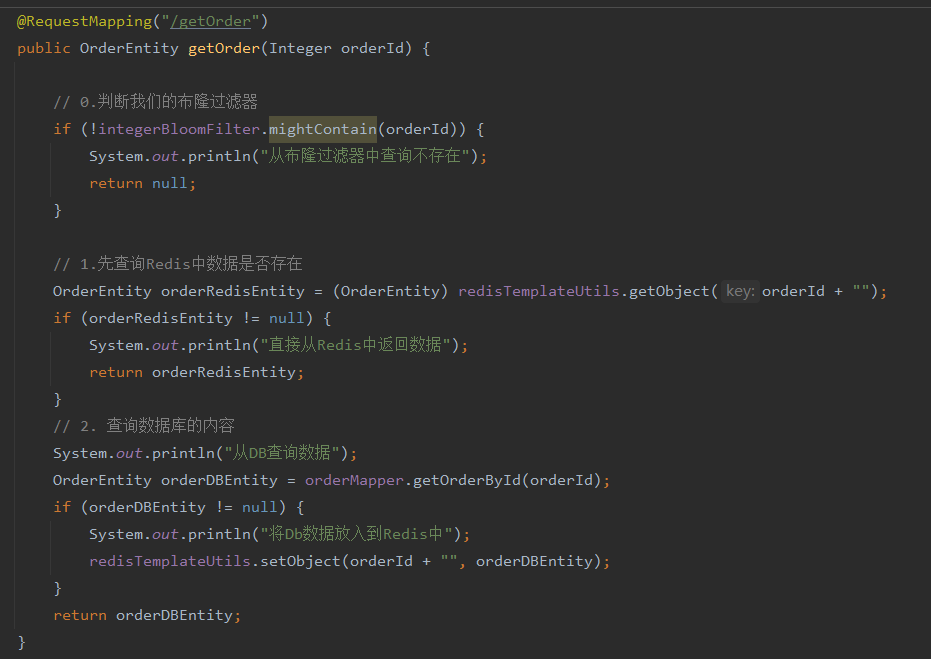


## 实际工作使用流程

1. 提前从数据库中预热数据



1. 查询Redis之前先进行布隆过滤。如果元素不存在在集合中，则终止业务流程

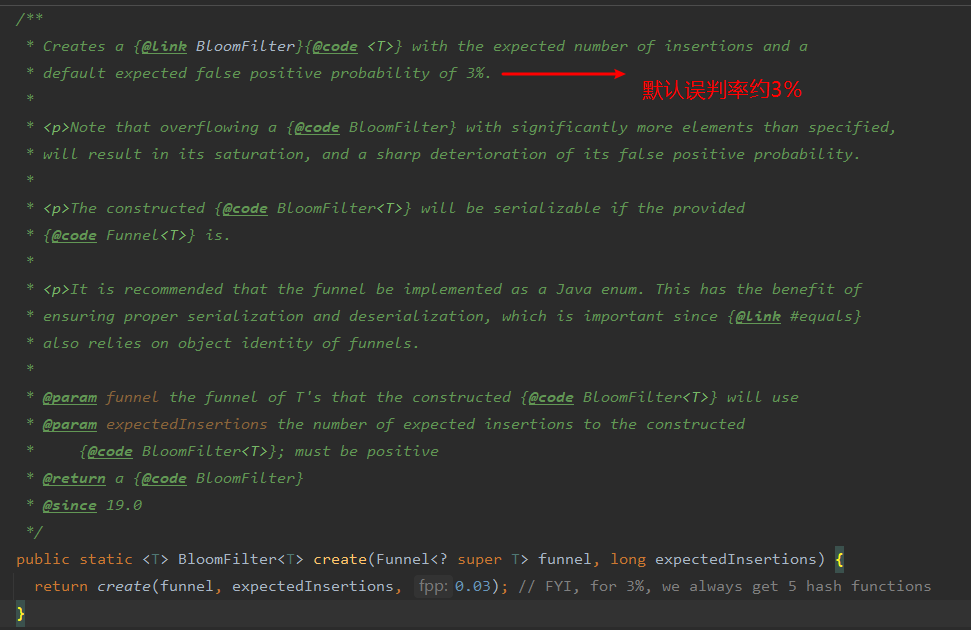


1. 当插入DB时，往布隆过滤器中添加元素。

## 测试布隆过滤器的效果

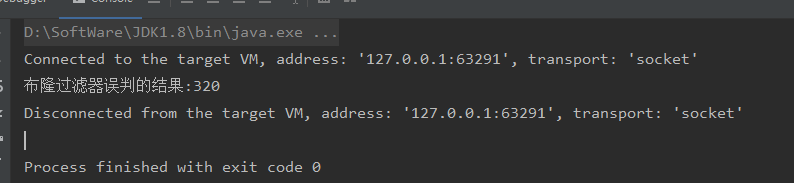
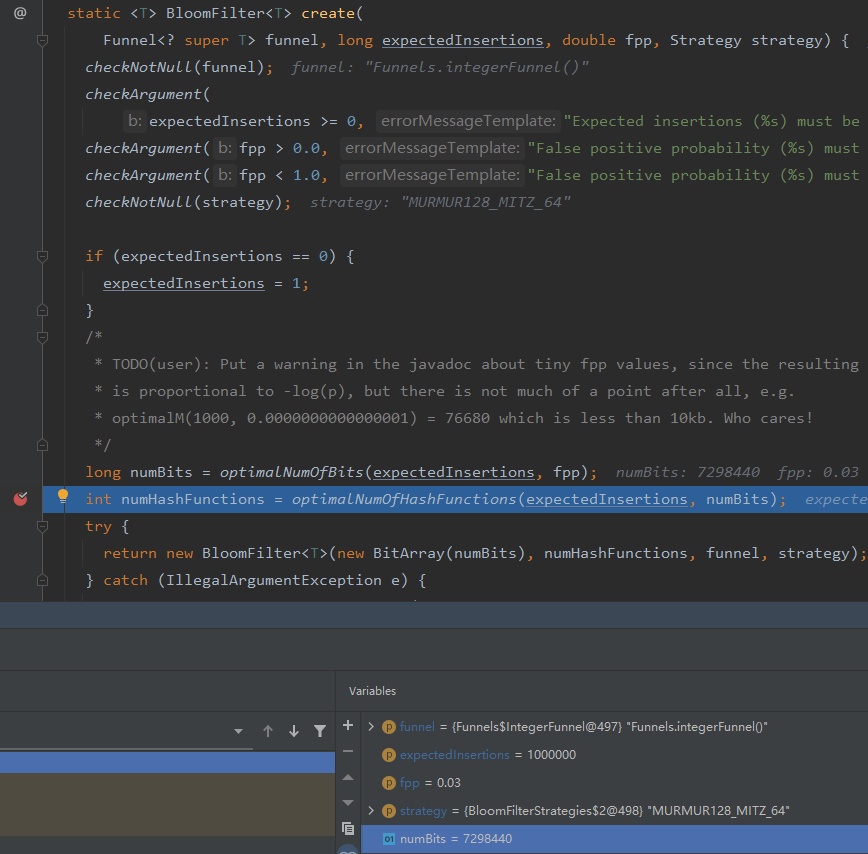
该案例往布隆过滤器插入100万条数据，并插入10000条不存在的数据，通过调整布隆过滤器的误判率，查看实际的效果

|  |
| --- |
| <**dependency**>  <**groupId**>com.google.guava</**groupId**>  <**artifactId**>guava</**artifactId**>  <**version**>22.0</**version**> </**dependency**>  public class BlongTest {   */\*\*  \* 假设集合中存放100万条数据  \*/* private static Integer *size* = 1000000;   public static void main(String[] args) {  BloomFilter<Integer> integerBloomFilter = BloomFilter.*create*(Funnels.*integerFunnel*(), *size*, 0.0000000001);  for (int i = 0; i < *size*; i++) {  // 向我们布隆过滤器中存放100万条数据  integerBloomFilter.put(i);  }  ArrayList<Integer> integers = new ArrayList<>();  for (int j = *size*; j < *size* + 10000; j++) {  // 使用该pai判断key在布隆过滤器中是否存在 返回true 存在 false 表示不存在  if (integerBloomFilter.mightContain(j)) {  // 将布隆过滤器误判的结果存放到集合中方便后期统计  integers.add(j);  }  }  System.*out*.println("布隆过滤器误判的结果:" + integers.size());  // 0.03概率 数组长度730万左右 0.01  // 0.01概率 数组长度960万左右  } } |

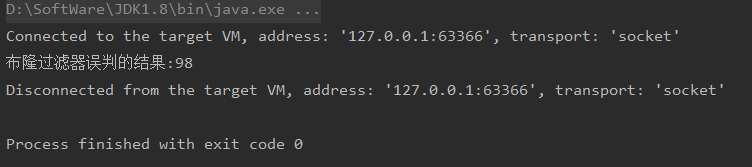
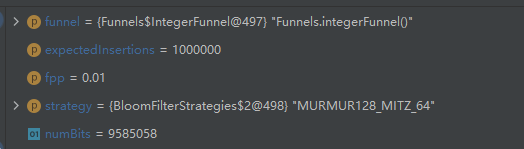


效果

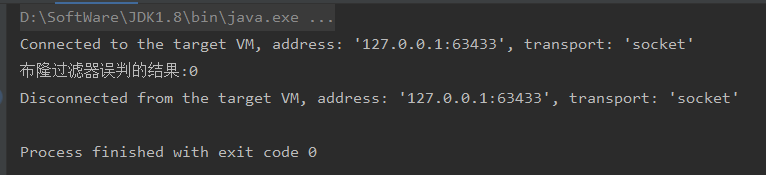
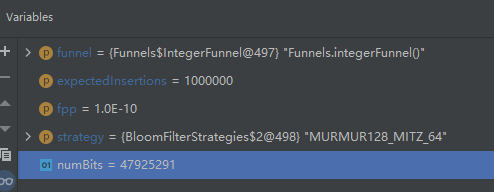
* 设置误判率为3%，位数组长度为730万，该案例中误判率为为3%



* 设置误判率为1%，位数组长度950万，该案例中误判率为1%



* 设置误判率极小0.00000001%，位数组长度4800万，该案例中误判率为0%



## 测试基于布隆过滤器解决项目中缓存穿透问题

|  |
| --- |
| @RestController public class OrderController {   @Autowired  private OrderMapper orderMapper;   @Autowired  private RedisUtils redisUtils;   @Autowired  private RedisTemplateUtils redisTemplateUtils;  BloomFilter<Integer> integerBloomFilter;   @RequestMapping("/addOrder")  public String addOrder() {  // 1.提前生成订单token 临时且唯一  String orderToken = UUID.*randomUUID*().toString();  Long orderId = System.*currentTimeMillis*();  // 2.将我们的token存放到rdis中  redisUtils.setString(orderToken, orderId + "", 10L);  OrderEntity orderEntity = new OrderEntity(null, "蚂蚁课堂永久会员", orderId + "", orderToken);  return orderMapper.insertOrder(orderEntity) > 0 ? "success" : "fail";  }   @RequestMapping("/getOrder")  public OrderEntity getOrder(Integer orderId) {   // 0.判断我们的布隆过滤器  if (!integerBloomFilter.mightContain(orderId)) {  System.*out*.println("从布隆过滤器中查询不存在");  return null;  }   // 1.先查询Redis中数据是否存在  OrderEntity orderRedisEntity = (OrderEntity) redisTemplateUtils.getObject(orderId + "");  if (orderRedisEntity != null) {  System.*out*.println("直接从Redis中返回数据");  return orderRedisEntity;  }  // 2. 查询数据库的内容  System.*out*.println("从DB查询数据");  OrderEntity orderDBEntity = orderMapper.getOrderById(orderId);  if (orderDBEntity != null) {  System.*out*.println("将Db数据放入到Redis中");  redisTemplateUtils.setObject(orderId + "", orderDBEntity);  }  return orderDBEntity;  }   @RequestMapping("/dbToBulong")  public String dbToBulong() {  // 1.从数据库预热id到布隆过滤器中  List<Integer> orderIds = orderMapper.getOrderIds();  integerBloomFilter = BloomFilter.*create*(Funnels.*integerFunnel*(), orderIds.size(), 0.01);  for (int i = 0; i < orderIds.size(); i++) {  // 添加到我们的布隆过滤器中  integerBloomFilter.put(orderIds.get(i));  }   return "success";  } } |

### 步骤

* 缓存预热

|  |
| --- |
| http://localhost:8080/dbToBulong |

* 访问不存在的Key

|  |
| --- |
| <http://localhost:8080/getOrder?orderId=1001>  <http://localhost:8080/getOrder?orderId=1002>  <http://localhost:8080/getOrder?orderId=1003>  <http://localhost:8080/getOrder?orderId=1004>  <http://localhost:8080/getOrder?orderId=1005> |

* 访问存在的key

|  |
| --- |
| http://localhost:8080/getOrder?orderId=21 |

## 布隆过滤器的一些问题

* 单个布隆过滤器只支持添加元素，不支持删除元素

|  |
| --- |
| https://en.wikipedia.org/wiki/Bloom\_filter  从这个简单的布隆过滤器中删除一个元素是不可能的，因为无法知道应该清除映射到的k位中的哪一个。尽管将这k个位中的任何一个设置为零就足以删除该元素，但它也将删除碰巧映射到该位的任何其他元素，因此清除任何位都可能会带来假阴性的可能性。  Removing an element from this simple Bloom filter is impossible because there is no way to tell which of the *k* bits it maps to should be cleared. Although setting any one of those *k* bits to zero suffices to remove the element, it would also remove any other elements that happen to map onto that bit. Since the simple algorithm provides no way to determine whether any other elements have been added that affect the bits for the element to be removed, clearing any of the bits would introduce the possibility of false negatives.  可以通过使第二个Bloom过滤器包含已删除的项目来模拟从Bloom过滤器中一次删除元素。然而，第二个过滤器中的假阳性变为复合过滤器中的假阴性()，这可能是不希望的。在这种方法中，不可能重新添加以前删除的项目，因为必须将其从“已删除”的过滤器中删除。  One-time removal of an element from a Bloom filter can be simulated by having a second Bloom filter that contains items that have been removed. However, false positives in the second filter become false negatives in the composite filter, which may be undesirable. In this approach re-adding a previously removed item is not possible, as one would have to remove it from the "removed" filter.  官网提出的另一种具体可参考方法是：布谷鸟过滤器 Cuckoo hashing |