# 20\_redis主从架构下如何才能做到99.99%的高可用性？

# 21\_redis哨兵架构的相关基础知识的讲解

# 22\_redis哨兵主备切换的数据丢失问题：异步复制、集群脑裂

# 23\_redis哨兵的多个核心底层原理的深入解析（包含slave选举算法）

# 24\_在项目中以经典的3节点方式部署哨兵集群

# 25\_对项目中的哨兵节点进行管理以及高可用redis集群的容灾演练

# 26\_redis如何在保持读写分离+高可用的架构下，还能横向扩容支撑1T+海量数据

# 27\_数据分布算法：hash+一致性hash+redis cluster的hash slot

# 28\_在项目中重新搭建一套读写分离+高可用+多master的redis cluster集群

# 29\_对项目的redis cluster实验多master写入、读写分离、高可用性

# 30\_redis cluster通过master水平扩容来支撑更高的读写吞吐+海量数据

# 31\_redis cluster的自动化slave迁移实现更强的高可用架构的部署方案

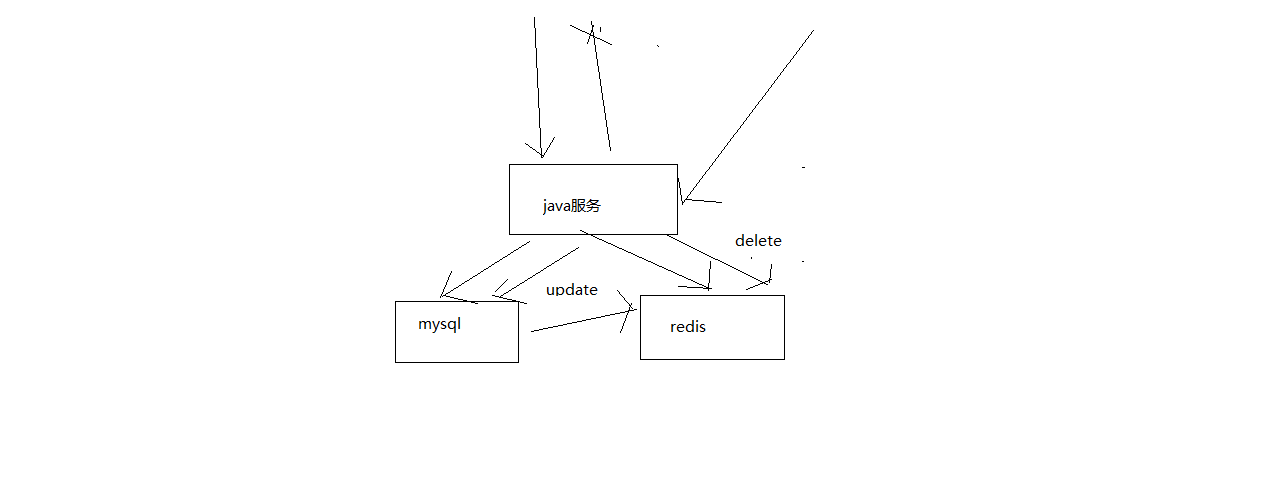
# 32\_redis cluster的核心原理分析：gossip通信、jedis smart定位、主备切换

# 33\_redis在实践中的一些常见问题以及优化思路（包含linux内核参数优化）

# 34\_redis阶段性总结：1T以上海量数据+10万以上QPS高并发+99.99%高可用

# 35\_亿级流量商品详情页的多级缓存架构以及架构中每一层的意义

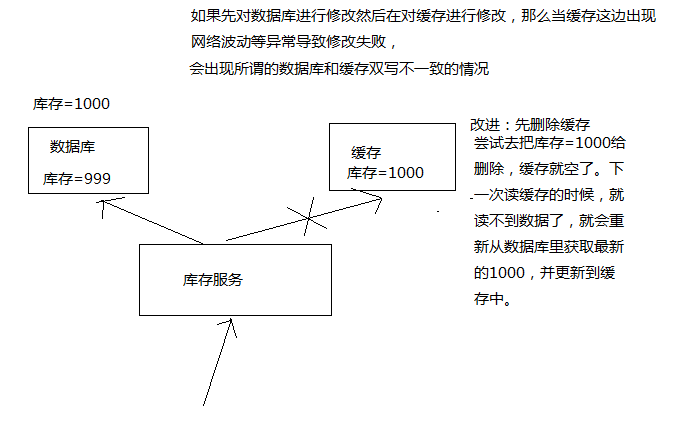
# 36\_Cache Aside Pattern缓存+数据库读写模式的分析



|  |
| --- |
| 最经典的缓存+数据库读写的模式，cache aside pattern  1、Cache Aside Pattern  （1）读的时候，先读缓存，缓存没有的话，那么就读数据库，然后取出数据后放入缓存，同时返回响应  （2）更新的时候，先删除缓存，然后再更新数据库  2、为什么是删除缓存，而不是更新缓存呢？  原因很简单，很多时候，复杂点的缓存的场景，因为缓存有的时候，不简单是数据库中直接取出来的值  商品详情页的系统，修改库存，只是修改了某个表的某些字段，但是要真正把这个影响的最终的库存计算出来，可能还需要从其他表查询一些数据，然后进行一些复杂的运算，才能最终计算出  现在最新的库存是多少，然后才能将库存更新到缓存中去  比如可能更新了某个表的一个字段，然后其对应的缓存，是需要查询另外两个表的数据，并进行运算，才能计算出缓存最新的值的  更新缓存的代价是很高的  是不是说，每次修改数据库的时候，都一定要将其对应的缓存去跟新一份？也许有的场景是这样的，但是对于比较复杂的缓存数据计算的场景，就不是这样了  如果你频繁修改一个缓存涉及的多个表，那么这个缓存会被频繁的更新，频繁的更新缓存  但是问题在于，这个缓存到底会不会被频繁访问到？？？  举个例子，一个缓存涉及的表的字段，在1分钟内就修改了20次，或者是100次，那么缓存跟新20次，100次; 但是这个缓存在1分钟内就被读取了1次，有大量的冷数据  28法则，黄金法则，20%的数据，占用了80%的访问量  实际上，如果你只是删除缓存的话，那么1分钟内，这个缓存不过就重新计算一次而已，开销大幅度降低  每次数据过来，就只是删除缓存，然后修改数据库，如果这个缓存，在1分钟内只是被访问了1次，那么只有那1次，缓存是要被重新计算的，用缓存才去算缓存  其实删除缓存，而不是更新缓存，就是一个lazy计算的思想，不要每次都重新做复杂的计算，不管它会不会用到，而是让它到需要被使用的时候再重新计算  mybatis，hibernate，懒加载，思想  查询一个部门，部门带了一个员工的list，没有必要说每次查询部门，都里面的1000个员工的数据也同时查出来啊  80%的情况，查这个部门，就只是要访问这个部门的信息就可以了  先查部门，同时要访问里面的员工，那么这个时候只有在你要访问里面的员工的时候，才会去数据库里面查询1000个员工 |

# 37\_高并发场景下的缓存+数据库双写不一致问题分析与解决方案设计

场景一：最初级的数据库+缓存双写不一致问题



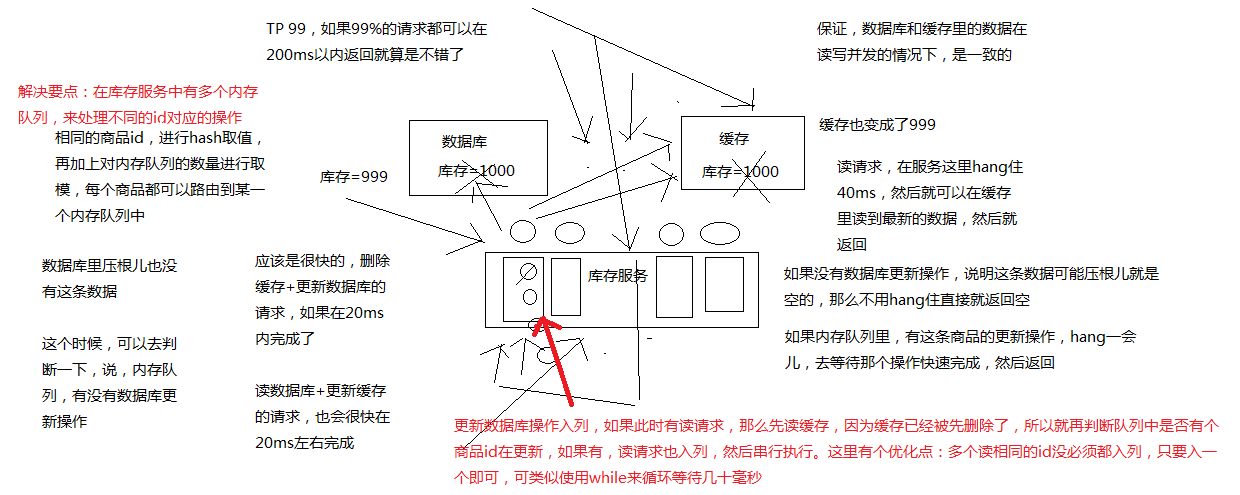
|  |
| --- |
| 1、最初级的缓存不一致问题以及解决方案  问题：先修改数据库，再删除缓存，如果删除缓存失败了，那么会导致数据库中是新数据，缓存中是旧数据，数据出现不一致  解决思路  先删除缓存，再修改数据库，如果删除缓存成功了，如果修改数据库失败了，那么数据库中是旧数据，缓存中是空的，那么数据不会不一致  因为读的时候缓存没有，则读数据库中旧数据，然后更新到缓存中 |

场景二：读写并发的时候复杂的数据库+缓存双写不一致的场景

先删缓存但高并发下依旧会出现数据不一致问题。

|  |
| --- |
| 2、比较复杂的数据不一致问题分析  数据发生了变更，先删除了缓存，然后要去修改数据库，此时还没修改成功，高并发下一个请求过来，去读缓存，发现缓存空了，去查询数据库，查到了修改前的旧数据，放到了缓存中  数据变更的程序完成了数据库的修改  完了，数据库和缓存中的数据不一样了。。。。  3、为什么上亿流量高并发场景下，缓存会出现这个问题？  只有在对一个数据在并发的进行读写的时候，才可能会出现这种问题  其实如果说你的并发量很低的话，特别是读并发很低，每天访问量就1万次，那么很少的情况下，会出现刚才描述的那种不一致的场景。但是问题是，如果每天的是上亿的流量，每秒并发读是几万，每秒只要有数据更新的请求，就可能会出现上述的数据库+缓存不一致的情况  高并发了以后，问题是很多的 |

解决办法：



|  |
| --- |
| 4、高并发下解决办法：数据库与缓存更新与读取操作进行异步串行化  更新数据的时候，根据数据的唯一标识，将操作路由之后，发送到一个jvm内部的队列中  读取数据的时候，如果发现数据不在缓存中，那么将重新读取数据+更新缓存的操作，根据唯一标识路由之后，也发送同一个jvm内部的队列中  一个队列对应一个工作线程  每个工作线程串行拿到对应的操作，然后一条一条的执行  这样的话，一个数据变更的操作，先执行，删除缓存，然后再去更新数据库，但是还没完成更新  此时如果一个读请求过来，读到了空的缓存，那么可以先将缓存更新的请求发送到队列中，此时会在队列中积压，然后同步等待缓存更新完成  这里有一个优化点，一个队列中，其实多个更新缓存请求串在一起是没意义的，因此可以做过滤，如果发现队列中已经有一个更新缓存的请求了，那么就不用再放个更新请求操作进去了，直接等待前面的更新操作请求完成即可  待那个队列对应的工作线程完成了上一个操作的数据库的修改之后，才会去执行下一个操作，也就是缓存更新的操作，此时会从数据库中读取最新的值，然后写入缓存中  如果请求还在等待时间范围内，不断轮询发现可以取到值了，那么就直接返回; 如果请求等待的时间超过一定时长，那么这一次直接从数据库中读取当前的旧值  5、高并发的场景下，该解决方案要注意的问题  （1）读请求长时阻塞  由于读请求进行了非常轻度的异步化，所以一定要注意读超时的问题，每个读请求必须在超时时间范围内返回  该解决方案，最大的风险点在于说，可能数据更新很频繁，导致队列中积压了大量更新操作在里面，然后读请求会发生大量的超时，最后导致大量的请求直接走数据库  务必通过一些模拟真实的测试，看看更新数据的频繁是怎样的  另外一点，因为一个队列中，可能会积压针对多个数据项的更新操作，因此需要根据自己的业务情况进行测试，可能需要部署多个服务，每个服务分摊一些数据的更新操作  如果一个内存队列里居然会挤压100个商品的库存修改操作，每隔库存修改操作要耗费10ms区完成，那么最后一个商品的读请求，可能等待10 \* 100 = 1000ms = 1s后，才能得到数据  这个时候就导致读请求的长时阻塞  一定要做根据实际业务系统的运行情况，去进行一些压力测试，和模拟线上环境，去看看最繁忙的时候，内存队列可能会挤压多少更新操作，可能会导致最后一个更新操作对应的读请求，会hang多少时间，如果读请求在200ms返回，如果你计算过后，哪怕是最繁忙的时候，积压10个更新操作，最多等待200ms，那还可以的  如果一个内存队列可能积压的更新操作特别多，那么你就要加机器，让每个机器上部署的服务实例处理更少的数据，那么每个内存队列中积压的更新操作就会越少  其实根据之前的项目经验，一般来说数据的写频率是很低的，因此实际上正常来说，在队列中积压的更新操作应该是很少的  针对读高并发，读缓存架构的项目，一般写请求相对读来说，是非常非常少的，每秒的QPS能到几百就不错了  一秒，500的写操作，5份，每200ms，就100个写操作  单机器，20个内存队列，每个内存队列，可能就积压5个写操作，每个写操作性能测试后，一般在20ms左右就完成  那么针对每个内存队列中的数据的读请求，也就最多hang一会儿，200ms以内肯定能返回了  写QPS扩大10倍，但是经过刚才的测算，就知道，单机支撑写QPS几百没问题，那么就扩容机器，扩容10倍的机器，10台机器，每个机器20个队列，200个队列  大部分的情况下，应该是这样的，大量的读请求过来，都是直接走缓存取到数据的  少量情况下，可能遇到读跟数据更新冲突的情况，如上所述，那么此时更新操作如果先入队列，之后可能会瞬间来了对这个数据大量的读请求，但是因为做了去重的优化，所以也就一个更新缓存的操作跟在它后面  等数据更新完了，读请求触发的缓存更新操作也完成，然后临时等待的读请求全部可以读到缓存中的数据  （2）读请求并发量过高  这里还必须做好压力测试，确保恰巧碰上上述情况的时候，还有一个风险，就是突然间大量读请求会在几十毫秒的延时hang在服务上，看服务能不能抗的住，需要多少机器才能抗住最大的极限情况的峰值  但是因为并不是所有的数据都在同一时间更新，缓存也不会同一时间失效，所以每次可能也就是少数数据的缓存失效了，然后那些数据对应的读请求过来，并发量应该也不会特别大  按1:99的比例计算读和写的请求，每秒5万的读QPS，可能只有500次更新操作  如果一秒有500的写QPS，那么要测算好，可能写操作影响的数据有500条，这500条数据在缓存中失效后，可能导致多少读请求，发送读请求到库存服务来，要求更新缓存  一般来说，1:1，1:2，1:3，每秒钟有1000个读请求，会hang在库存服务上，每个读请求最多hang多少时间，200ms就会返回  在同一时间最多hang住的可能也就是单机200个读请求，同时hang住  单机hang200个读请求，还是ok的  1:20，每秒更新500条数据，这500秒数据对应的读请求，会有20 \* 500 = 1万  1万个读请求全部hang在库存服务上，就死定了 |

**由上面的解决方案引发的问题：**

同一商品id请求没发到同一台服务器上、没发送到同一机器的相同内存队列

|  |
| --- |
| （3）多服务实例部署的请求路由  可能这个服务部署了多个实例，那么必须保证说，执行数据更新操作，以及执行缓存更新操作的请求，都通过nginx服务器路由到相同的服务实例上。  可以做服务间的按照某个请求参数的hash路由、nginx、hash路由的功能来解决。  （4）热点商品的路由问题，导致请求的倾斜  万一某个商品的读写请求特别高，全部打到相同的机器的相同的队列里面去了，可能造成某台机器的压力过大。就是说，因为只有在商品数据更新的时候才会清空缓存，然后才会导致读写并发，所以更新频率不是太高的话，这个问题的影响并不是特别大。但是的确可能某些机器的负载会高一些。 |

# 38\_在linux虚拟机中安装部署MySQL数据库

# 39\_库存服务的开发框架整合与搭建：spring boot+mybatis+jedis

|  |
| --- |
| 1、pom.xml  maven下载这些依赖的时候，会非常非常的慢，根据不同人的网络环境，也不一样  你就在eclipse里面，观察每个依赖的下载的情况  如果说觉得下载的很慢，就是卡在一个地方，好长时间不能下载，进度条都不动了  那就手动下载maven依赖，mysql connector java maven，进到maven中央依赖库里面，去手动下载对应版本的jar包，可以用迅雷，会比较快速一些  你需要手动将jar包拷贝到你本地的maven仓库的对应的目录中去  可能对应的目录不存在，那就自己手动创建; 可能对应的目录已经存在，那么需要你将里面的东西先删除，然后拷贝自己下载的jar包进去  然后需要执行mvn install命令，手动安装依赖  mvn install:install-file -Dfile=F:\apache-maven-3.0.5\mvn\_repo\redis\clients\jedis\2.5.2\jedis-2.5.2.jar -DgroupId=redis.clients -DartifactId=jedis -Dversion=2.5.2 -Dpackaging=jar  可能需要将eclipse强制关闭，任务管理器里面，直接强制结束任务，关闭eclipse，重新打开  右击工程，强制重新更新maven的依赖  然后就会继续去下载接下来的依赖包  我大概手动下载了将近10个依赖，然后才顺利的下载完了所有的依赖  <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.2.5.RELEASE</version>  </parent>  <properties>  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  <java.version>1.8</java.version>  </properties>  <dependencies>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-thymeleaf</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-jdbc</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-actuator</artifactId>  </dependency>  <dependency>  <groupId>org.mybatis</groupId>  <artifactId>mybatis-spring</artifactId>  <version>1.2.2</version>  </dependency>  <dependency>  <groupId>org.mybatis</groupId>  <artifactId>mybatis</artifactId>  <version>3.2.8</version>  </dependency>  <dependency>  <groupId>org.apache.tomcat</groupId>  <artifactId>tomcat-jdbc</artifactId>  </dependency>  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  </dependency>  <dependency>  <groupId>com.alibaba</groupId>  <artifactId>fastjson</artifactId>  <version>1.1.43</version>  </dependency>  </dependencies>  <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  </plugins>  </build>  <repositories>  <repository>  <id>spring-milestone</id>  <url>https://repo.spring.io/libs-release</url>  </repository>  </repositories>  <pluginRepositories>  <pluginRepository>  <id>spring-milestone</id>  <url>https://repo.spring.io/libs-release</url>  </pluginRepository>  </pluginRepositories>  2、com.roncoo.eshop.inventory.Application  我们是直接讲解项目的，项目中遇到的一些技术，比如说redis，是缓存架构topic里面非常重要的一个环节，那我们当然会花费很大的篇幅去仔细讲解  但是比如spring boot、mybatis、zookeeper、storm  有些技术，是默认大家要会的，spring boot，mybatis，你说你都不会，不行，自己去学一下  zookeeper，kafka，技术，直接带着大家部署，简单介绍一下，就开始用了，如果真的是要用这个技术，做项目，还是得上网自己去查阅一些资料，学习这个技术  storm，会带着大家把20%的核心基础知识学习一下，作为java程序员，可以开发就行了  一个项目课程，涉及了十几种技术，我不可能全都给你按照从入门到精通的方式，全部讲解一遍  @EnableAutoConfiguration  @SpringBootApplication  @ComponentScan  @MapperScan("com.roncoo.eshop.inventory.mapper")  public class Application {    @Bean  @ConfigurationProperties(prefix="spring.datasource")  public DataSource dataSource() {  return new org.apache.tomcat.jdbc.pool.DataSource();  }    @Bean // mybatis的SqlSessionFactory  public SqlSessionFactory sqlSessionFactoryBean() throws Exception {  SqlSessionFactoryBean sqlSessionFactoryBean = new SqlSessionFactoryBean();  sqlSessionFactoryBean.setDataSource(dataSource());  PathMatchingResourcePatternResolver resolver = new PathMatchingResourcePatternResolver();  sqlSessionFactoryBean.setMapperLocations(resolver.getResources("classpath:/mybatis/\*.xml"));  return sqlSessionFactoryBean.getObject();  }    @Bean // 构建事物管理器  public PlatformTransactionManager transactionManager() {  return new DataSourceTransactionManager(dataSource());  }  public static void main(String[] args) {  SpringApplication.run(Application.class, args);  }    }  3、com.roncoo.eshop.inventory.model.User  4、com.roncoo.eshop.inventory.mapper.UserMapper  public interface UserMapper {  public User findUserInfo();  }  5、com.roncoo.eshop.inventory.service.UserService  6、UserController  @Controller  public class UserController {    @Autowired  private UserService userService;    @RequestMapping("/getUserInfo")  @ResponseBody  public User getUserInfo() {  User user = userService.getUserInfo();  return user;  }  }  7、resources/Application.properties  spring.datasource.url=jdbc:mysql://127.0.0.1:3306/test  spring.datasource.username=root  spring.datasource.password=root  spring.datasource.driver-class-name=com.mysql.jdbc.Driver  8、resources/mybatis/UserMapper.xml  <?xml version="1.0" encoding="UTF-8"?>  <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  <mapper namespace="com.roncoo.eshop.inventory.mapper.UserMapper">    <select id="findUserInfo" resultType="com.roncoo.eshop.inventory.model.User">  select name,age from user;  </select>    </mapper>  9、测试spring boot+mybatis的整合  在数据库中创建一个eshop database，创建一个eshop账号和密码，创建一个user表，里面插入一条数据，张三和25岁  create database if not exists eshop;  grant all privileges on eshop.\* to 'eshop'@'%' identified by 'eshop';  create table user(name varchar(255), age int)  insert into user values('张三', 25)  启动Application程序，访问getUserInfo接口，能否从mysql中查询数据，并返回到页面上  如果可以，说明spring boot+mybatis整合成功  10、整合Jedis Cluster  <dependency>  <groupId>redis.clients</groupId>  <artifactId>jedis</artifactId>  </dependency>  Application  @Bean  public JedisCluster JedisClusterFactory() {  Set<HostAndPort> jedisClusterNodes = new HashSet<HostAndPort>();  jedisClusterNodes.add(new HostAndPort("192.168.31.19", 7003));  jedisClusterNodes.add(new HostAndPort("192.168.31.19", 7004));  jedisClusterNodes.add(new HostAndPort("192.168.31.227", 7006));  JedisCluster jedisCluster = new JedisCluster(jedisClusterNodes);  return jedisCluster;  }  @Repository("redisDAO")  public class RedisDAOImpl implements RedisDAO {  @Resource  private JedisCluster jedisCluster;    @Override  public void set(String key, String value) {  jedisCluster.set(key, value);  }  @Override  public String get(String key) {  return jedisCluster.get(key);  }  }  UserServiceImpl  @Override  public User getCachedUserInfo() {  redisDAO.set("cached\_user", "{\"name\": \"zhangsan\", \"age\": 25}") ;  String json = redisDAO.get("cached\_user");  JSONObject jsonObject = JSONObject.parseObject(json);    User user = new User();  user.setName(jsonObject.getString("name"));  user.setAge(jsonObject.getInteger("age"));    return user;  }  UserController  @RequestMapping("/getCachedUserInfo")  @ResponseBody  public User getCachedUserInfo() {  User user = userService.getCachedUserInfo();  return user;  } |

# 40\_在库存服务中实现缓存与数据库双写一致性保障方案（一）

|  |
| --- |
| 更新数据的时候，根据数据的唯一标识，将操作路由之后，发送到一个jvm内部的队列中  读取数据的时候，如果发现数据不在缓存中，那么将重新读取数据+更新缓存的操作，根据唯一标识路由之后，也发送同一个jvm内部的队列中  一个队列对应一个工作线程  每个工作线程串行拿到对应的操作，然后一条一条的执行  这样的话，一个数据变更的操作，先执行，删除缓存，然后再去更新数据库，但是还没完成更新  此时如果一个读请求过来，读到了空的缓存，那么可以先将缓存更新的请求发送到队列中，此时会在队列中积压，然后同步等待缓存更新完成  这里有一个优化点，一个队列中，其实多个更新缓存请求串在一起是没意义的，因此可以做过滤，如果发现队列中已经有一个更新缓存的请求了，那么就不用再放个更新请求操作进去了，直接等待前面的更新操作请求完成即可  待那个队列对应的工作线程完成了上一个操作的数据库的修改之后，才会去执行下一个操作，也就是缓存更新的操作，此时会从数据库中读取最新的值，然后写入缓存中  如果请求还在等待时间范围内，不断轮询发现可以取到值了，那么就直接返回; 如果请求等待的时间超过一定时长，那么这一次直接从数据库中读取当前的旧值  int h;  return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);  (queueNum - 1) & hash  1、线程池+内存队列初始化  @Bean  public ServletListenerRegistrationBean servletListenerRegistrationBean(){  ServletListenerRegistrationBean servletListenerRegistrationBean = new ServletListenerRegistrationBean();  servletListenerRegistrationBean.setListener(new InitListener());  return servletListenerRegistrationBean;  }  java web应用，做系统的初始化，一般在哪里做呢？  ServletContextListener里面做，listener，会跟着整个web应用的启动，就初始化，类似于线程池初始化的构建  spring boot应用，Application，搞一个listener的注册  2、两种请求对象封装  3、请求异步执行Service封装  4、两种请求Controller接口封装  5、读请求去重优化  6、空数据读请求过滤优化  空数据读请求过滤优化解释：（就是上一节双写不一致中优化的点）  队列  对一个商品的库存的数据库更新操作已经在内存队列中了  然后对这个商品的库存的读取操作，要求读取数据库的库存数据，然后更新到缓存中，多个读  这多个读，其实只要有一个读请求操作压到队列里就可以了  其他的读操作，全部都wait那个读请求的操作，刷新缓存，就可以读到缓存中的最新数据了  如果读请求发现redis缓存中没有数据，就会发送读请求给库存服务，但是此时缓存中为空，可能是因为写请求先删除了缓存，也可能是数据库里压根儿没这条数据  如果是数据库中压根儿没这条数据的场景，那么就不应该将读请求操作给压入队列中，而是直接返回空就可以了  都是为了减少内存队列中的请求积压，内存队列中积压的请求越多，就可能导致每个读请求hang住的时间越长，也可能导致多个读请求被hang住 |

# 41\_在库存服务中实现缓存与数据库双写一致性保障方案（二）

# 42\_在库存服务中实现缓存与数据库双写一致性保障方案（三）

# 43\_在库存服务中实现缓存与数据库双写一致性保障方案（四）

# 44\_库存服务代码调试以及打印日志观察服务的运行流程是否正确

# 45\_商品详情页结构分析、缓存全量更新问题以及缓存维度化解决方案

# 46\_缓存数据生产服务的工作流程分析以及工程环境搭建

# 47\_完成spring boot整合ehcache的搭建以支持服务本地堆缓存

# 48\_redis的LRU缓存清除算法讲解以及相关配置使用

|  |
| --- |
| 之前给大家讲解过，多级缓存架构，缓存数据生产服务，监听各个数据源服务的数据变更的消息，得到消息之后，然后调用接口拉去数据  将拉去到的数据，写入本地ehcache缓存一份，spring boot整合，演示过  数据写入redis分布式缓存中一份，你不断的将数据写入redis，写入redis，然后redis的内存是有限的，每个redis实例最大一般也就是设置给10G  那如果你不断的写入数据，当数据写入的量超过了redis能承受的范围之后，改该怎么玩儿呢？？？  redis是会在数据达到一定程度之后，超过了一个最大的限度之后，就会将数据进行一定的清理，从内存中清理掉一些数据  只有清理掉一些数据之后，才能将新的数据写入内存中  1、LRU算法概述  redis默认情况下就是使用LRU策略的，因为内存是有限的，但是如果你不断地往redis里面写入数据，那肯定是没法存放下所有的数据在内存的  所以redis默认情况下，当内存中写入的数据很满之后，就会使用LRU算法清理掉部分内存中的数据，腾出一些空间来，然后让新的数据写入redis缓存中  LRU：Least Recently Used，最近最少使用算法  将最近一段时间内，最少使用的一些数据，给干掉。比如说有一个key，在最近1个小时内，只被访问了一次; 还有一个key在最近1个小时内，被访问了1万次  这个时候比如你要将部分数据给清理掉，你会选择清理哪些数据啊？肯定是那个在最近小时内被访问了1万次的数据  2、缓存清理设置  redis.conf  maxmemory，设置redis用来存放数据的最大的内存大小，一旦超出这个内存大小之后，就会立即使用LRU算法清理掉部分数据  如果用LRU，那么就是将最近最少使用的数据从缓存中清除出去  对于64 bit的机器，如果maxmemory设置为0，那么就默认不限制内存的使用，直到耗尽机器中所有的内存为止; 但是对于32 bit的机器，有一个隐式的闲置就是3GB  maxmemory-policy，可以设置内存达到最大闲置后，采取什么策略来处理  （1）noeviction: 如果内存使用达到了maxmemory，client还要继续写入数据，那么就直接报错给客户端  （2）allkeys-lru: 就是我们常说的LRU算法，移除掉最近最少使用的那些keys对应的数据  （3）volatile-lru: 也是采取LRU算法，但是仅仅针对那些设置了指定存活时间（TTL）的key才会清理掉  （4）allkeys-random: 随机选择一些key来删除掉  （5）volatile-random: 随机选择一些设置了TTL的key来删除掉  （6）volatile-ttl: 移除掉部分keys，选择那些TTL时间比较短的keys  在redis里面，写入key-value对的时候，是可以设置TTL，存活时间，比如你设置了60s。那么一个key-value对，在60s之后就会自动被删除  redis的使用，各种数据结构，list，set，等等  allkeys-lru  这边拓展一下思路，对技术的研究，一旦将一些技术研究的比较透彻之后，就喜欢横向对比底层的一些原理  storm，科普一下  玩儿大数据的人搞得，领域，实时计算领域，storm  storm有很多的流分组的一些策略，按shuffle分组，global全局分组，direct直接分组，fields按字段值hash后分组  分组策略也很多，但是，真正公司里99%的场景下，使用的也就是shuffle和fields，两种策略  redis，给了这么多种乱七八糟的缓存清理的算法，其实真正常用的可能也就那么一两种，allkeys-lru是最常用的  3、缓存清理的流程  （1）客户端执行数据写入操作  （2）redis server接收到写入操作之后，检查maxmemory的限制，如果超过了限制，那么就根据对应的policy清理掉部分数据  （3）写入操作完成执行  4、redis的LRU近似算法  科普一个相对来说稍微高级一丢丢的知识点  redis采取的是LRU近似算法，也就是对keys进行采样，然后在采样结果中进行数据清理  redis 3.0开始，在LRU近似算法中引入了pool机制，表现可以跟真正的LRU算法相当，但是还是有所差距的，不过这样可以减少内存的消耗  redis LRU算法，是采样之后再做LRU清理的，跟真正的、传统、全量的LRU算法是不太一样的  maxmemory-samples，比如5，可以设置采样的大小，如果设置为10，那么效果会更好，不过也会耗费更多的CPU资源 |

# 49\_zookeeper+kafka集群的安装部署以及如何简单使用的介绍

# 50\_基于kafka+ehcache+redis完成缓存数据生产服务的开发与测试

# 51\_基于“分发层+应用层”双层nginx架构提升缓存命中率方案分析

# 51\_基于“分发层+应用层”双层nginx架构提升缓存命中率方案分析.zip

# 52\_基于OpenResty部署应用层nginx以及nginx+lua开发hello world

# 52\_基于OpenResty部署应用层nginx以及nginx+lua开发hello world.zip

# 53\_部署分发层nginx以及基于lua完成基于商品id的定向流量分发策略

# 53\_部署分发层nginx以及基于lua完成基于商品id的定向流量分发策略.zip

# 54\_基于nginx+lua+java完成多级缓存架构的核心业务逻辑（一）

# 54\_基于nginx+lua+java完成多级缓存架构的核心业务逻辑（一）.zip

# 55\_基于nginx+lua+java完成多级缓存架构的核心业务逻辑（二）

# 55\_基于nginx+lua+java完成多级缓存架构的核心业务逻辑（二）.zip

# 56\_基于nginx+lua+java完成多级缓存架构的核心业务逻辑（三）

# 56\_基于nginx+lua+java完成多级缓存架构的核心业务逻辑（三）.zip

# 57\_分布式缓存重建并发冲突问题以及zookeeper分布式锁解决方案

# 57\_分布式缓存重建并发冲突问题以及zookeeper分布式锁解决方案.zip

# 58\_缓存数据生产服务中的zk分布式锁解决方案的代码实现（一）

# 58\_缓存数据生产服务中的zk分布式锁解决方案的代码实现（一）.zip

# 59\_缓存数据生产服务中的zk分布式锁解决方案的代码实现（二）

# 59\_缓存数据生产服务中的zk分布式锁解决方案的代码实现（二）.zip

# 60\_缓存数据生产服务中的zk分布式锁解决方案的代码实现（三）

# 60\_缓存数据生产服务中的zk分布式锁解决方案的代码实现（三）.zip

# 61\_Java程序员、缓存架构以及Storm大数据实时计算之间的关系.zip

# 62\_讲给Java工程师的史上最通俗易懂Storm教程：大白话介绍.zip

# 63\_讲给Java工程师的史上最通俗易懂Storm教程：大白话讲集群架构与核心概念.zip

# 64\_讲给Java工程师的史上最通俗易懂Storm教程：大白话讲并行度和流分组.zip

# 65\_讲给Java工程师的史上最通俗易懂Storm教程：纯手敲WordCount程序.zip

# 66\_讲给Java工程师的史上最通俗易懂Storm教程：纯手工集群部署.zip

# 67\_讲给Java工程师的史上最通俗易懂Storm教程：基于集群运行计算拓扑.zip

# 68\_缓存冷启动问题：新系统上线、redis彻底崩溃导致数据无法恢复.zip

# 69\_缓存预热解决方案：基于storm实时热点统计的分布式并行缓存预热.zip

# 70\_基于nginx+lua完成商品详情页访问流量实时上报kafka的开发.zip

# 71\_基于storm+kafka+LRUMap完成商品访问次数实时统计拓扑的开发.zip

# 72\_基于storm完成LRUMap中topn热门商品列表的算法讲解与编写.zip

# 73\_基于storm+zookeeper以及分布式锁完成热门商品列表的分段存储.zip

# 74\_基于双重zookeeper分布式锁完成分布式并行缓存预热的代码开发.zip

# 75\_将缓存预热解决方案的代码运行后观察效果以及调试和修复所有的bug.zip

# 76\_热点缓存问题：促销抢购时的超级热门商品可能导致系统全盘崩溃的场景.zip

# 77\_基于nginx+lua+storm的热点缓存的流量分发策略自动降级解决方案.zip

# 78\_在storm拓扑中加入热点缓存实时自动识别和感知的代码逻辑.zip

# 79\_在storm拓扑中加入nginx反向推送缓存热点与缓存数据的代码逻辑.zip

# 80\_在流量分发+后端应用双层nginx中加入接收热点缓存数据的接口.zip

# 81\_在nginx+lua中实现热点缓存自动降级为负载均衡流量分发策略的逻辑.zip

# 82\_在storm拓扑中加入热点缓存消失的实时自动识别和感知的代码逻辑.zip

# 83\_将热点缓存自动降级解决方案的代码运行后观察效果以及调试和修复bug.zip

# 84\_hystrix与高可用系统架构：资源隔离+限流+熔断+降级+运维监控

# 84\_hystrix与高可用系统架构：资源隔离+限流+熔断+降级+运维监控.zip

# 85\_hystrix要解决的分布式系统可用性问题以及其设计原则.zip

# 86\_电商网站的商品详情页缓存服务业务背景以及框架结构说明.zip

# 87\_基于spring boot快速构建缓存服务以及商品服务.zip

# 88\_快速完成缓存服务接收数据变更消息以及调用商品服务接口的代码编写.zip

# 89\_商品服务接口故障导致的高并发访问耗尽缓存服务资源的场景分析.zip

# 90\_基于hystrix的线程池隔离技术进行商品服务接口的资源隔离以及限流.zip

# 91\_基于hystrix的信号量技术对地理位置获取逻辑进行资源隔离与限流.zip

# 92\_hystrix的线程池+服务+接口划分以及资源池的容量大小控制.zip

# 93\_深入分析hystrix执行时的8大流程步骤以及内部原理.zip

# 94\_基于request cache请求缓存技术优化批量商品数据查询接口.zip

# 95\_开发品牌名称获取接口的基于本地缓存的fallback降级机制.zip

# 96\_深入理解hystrix的短路器原理以及接口异常时的熔断实验.zip

# 97\_深入理解线程池隔离技术的设计原则以及动手实战接口限流实验.zip

# 98\_基于timeout机制来为商品服务接口的调用超时提供安全保护.zip

# 99\_基于hystrix的高可用分布式系统架构项目实战课程的总结.zip