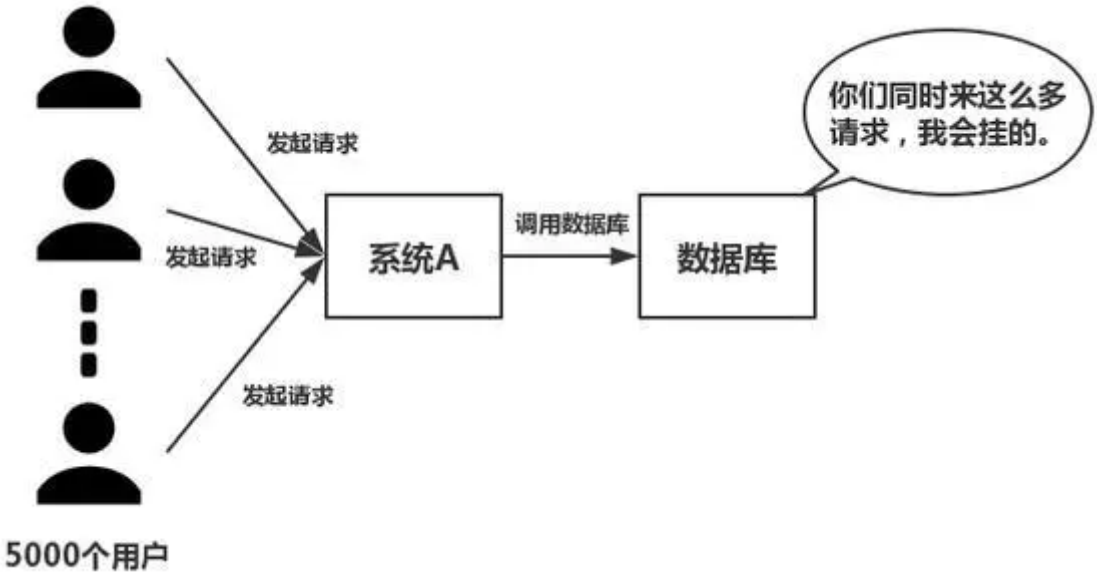
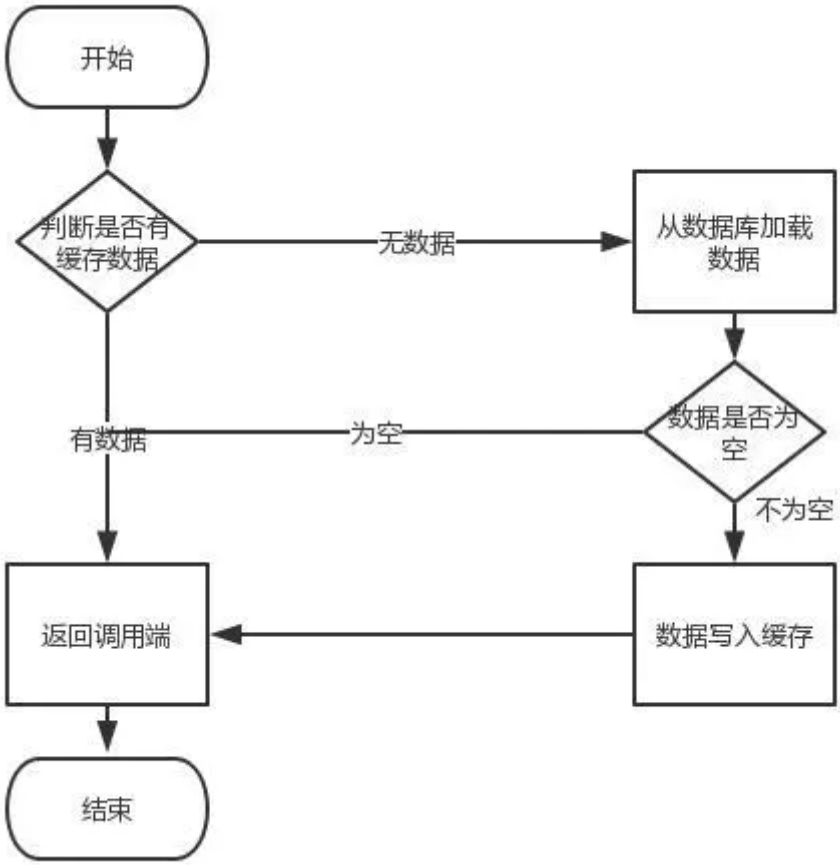
Redis缓存与数据库双写一致性解决方案

1、需求起因

在高并发的业务场景下，数据库大多数情况都是用户并发访问最薄弱的环节。所以，就需要使用redis做一个缓冲操作，让请求先访问到redis，而不是直接访问MySQL等数据库！



好了，我们现在引入缓存的概念，那么访问路程变成了如下：



上面这个经典的读取缓存步骤一般没有什么问题，但是一旦涉及到数据更新：**数据库和缓存更新**，就容易出现**缓存(Redis)和数据库间的数据一致性问题。**

有以下这些不一致的场景：

1.当更新数据时，如更新某商品的库存，当前商品的库存是100，现在要更新为99，先更新数据库更改成99，然后更新缓存，发现更新缓存失败了，这意味着数据库存的是99，而缓存还是100，这导致数据库和缓存不一致

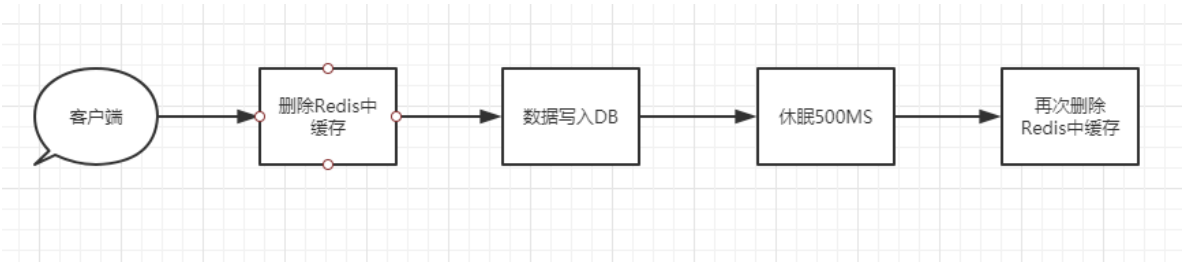
2.如果删除了缓存Redis记录，还没有来得及删除对应的数据库记录，另一个线程就来读取，发现缓存为空，则去数据库中读取数据写入缓存，此时缓存中为脏数据。

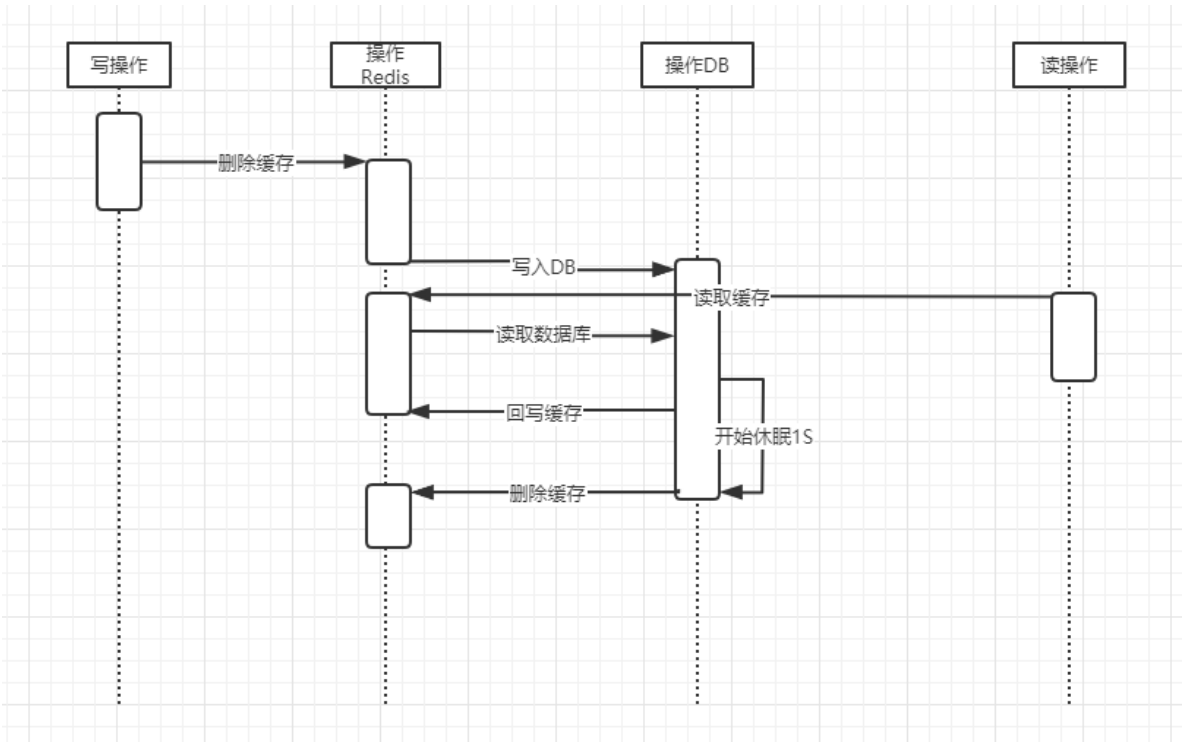
因为写和读是并发的，没法保证顺序,就会出现缓存和数据库的数据不一致的问题。

2、解决方案

2.1 延时双删策略

基本流程图：





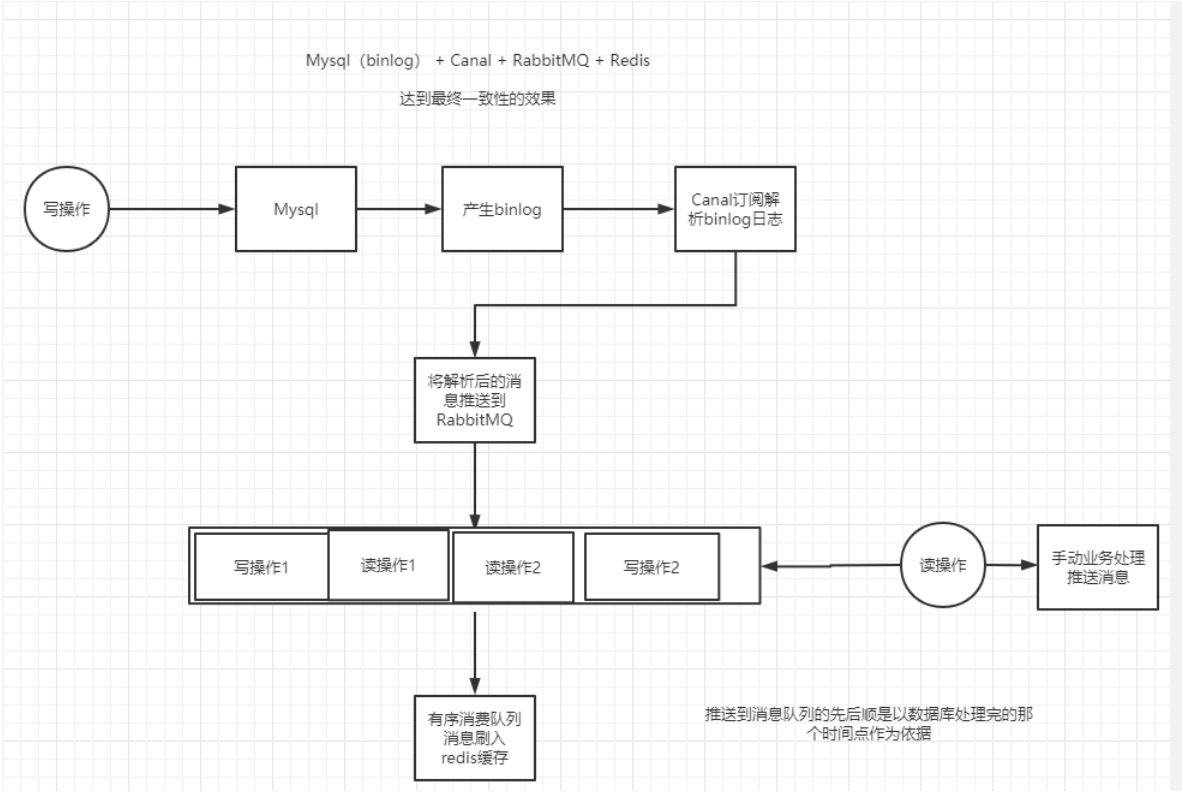
伪代码：

```
public void write(String key,Object data){

        redis.delKey(key);

        db.updateData(data);

        Thread.sleep(1000);
```

```
        redis.delKey(key);

    }

/*转化为中文描述就是
（1）先淘汰缓存
（2）再写数据库
（3）休眠1秒（根据具体的读操作业务时间来定）
（4）再次淘汰缓存

这么做，可以将1秒内所造成的缓存脏数据，再次删除。
那么，这个1秒怎么确定的，具体该休眠多久呢？
针对上面的情形，读者应该自行评估自己的项目的读数据业务逻辑的耗时。然后写数据的休眠时间则在读数据业务逻辑的耗时基础上，加几百ms即可。这么做的目的，就是确保读请求结束，写请求可以删除读请求造成的缓存脏数据。
如果你用了mysql的读写分离架构怎么办？
ok，在这种情况下，造成数据不一致的原因如下，还是两个请求，一个请求A进行更新操作，另一个请求B进行查询操作。
（1）请求A进行写操作，删除缓存
（2）请求A将数据写入数据库了，
（3）请求B查询缓存发现，缓存没有值
（4）请求B去从库查询，这时，还没有完成主从同步，因此查询到的是旧值
（5）请求B将旧值写入缓存
（6）数据库完成主从同步，从库变为新值
（7）请求A将缓存中B写入的旧值数据删除

上述情形，就是数据不一致的原因。还是使用双删延时策略。只是，睡眠时间修改为在主从同步的延时时间基础上，加几百ms。

采用这种同步淘汰策略，吞吐量降低怎么办？
ok，那就将第二次删除作为异步的。自己起一个线程，异步删除。这样，写的请求就不用沉睡一段时间后了，再返回。这么做，加大吞吐量。
第二次删除,如果删除失败怎么办？
这是个非常好的问题，因为第二次删除失败，就会出现如下情形。还是有两个请求，一个请求A进行更新操作，另一个请求B进行查询操作，为了方便，假设是单库：
（1）请求A进行写操作，删除缓存
（2）请求B查询发现缓存不存在
（3）请求B去数据库查询得到旧值
（4）请求B将旧值写入缓存
（5）请求A将新值写入数据库
（6）请求A试图去删除请求B写入的缓存值，结果失败了。

ok,这也就是说。如果第二次删除缓存失败，会再次出现缓存和数据库不一致的问题。咋办？
们需要提供一个保障重试的方案：
      定时任务，这样会压力太大，并且一直阻塞会影响性能
      消息队列，异步处理，可以让性能提升，但是对业务代码造成大量的侵入

最后的最后，这个延时的时间，你真的好把握吗？
```

2.2 异步更新缓存(基于订阅binlog的同步机制)

基本流程图:

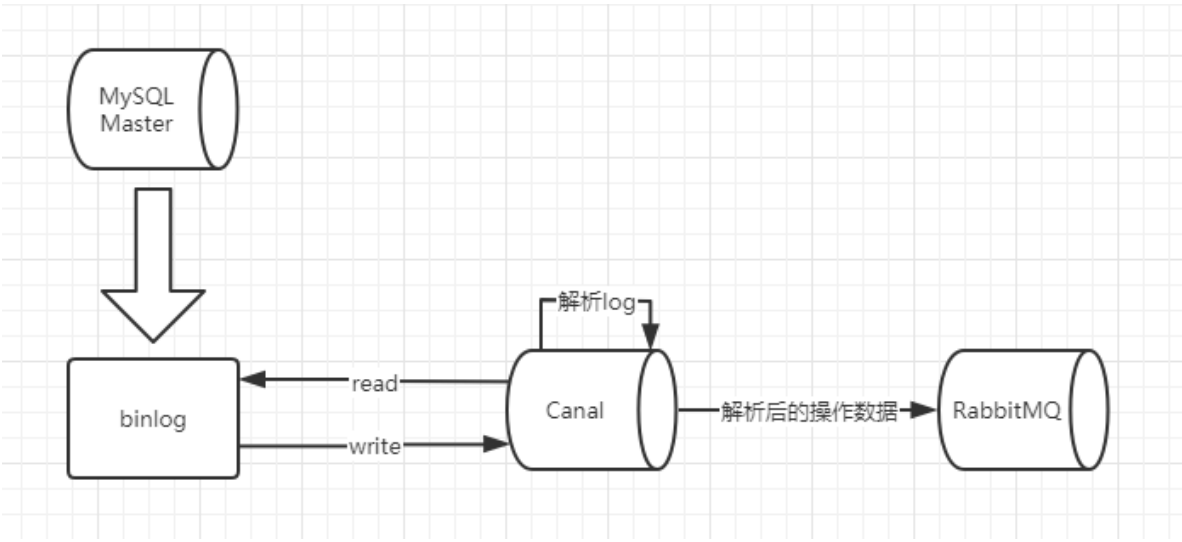


这个也是我们今天的主要讲解，也是业界用得最多的方案，该方案的核心在于**使用队列对读写操作进行排队操作**，保证了数据的最终一致性，当然，性能相对上一种要差，但是还是那句话，数据的准确性才是最重要的！

2.3 初识Canal

阿里Canal主要是听过伪装成mysql从节点来向主节点拉取binlog日志解析成消息推送到MQ消息队列。

Canal在双写一致性中所处的位置：



1、canal模拟mysql slave的交互协议，伪装自己为mysql slave，向mysql master发送dump协议
2、mysql master收到dump请求，开始推送binary log给slave(也就是canal)
3、canal解析binary log对象(原始为byte流)

4、canal将解析后的对象数据推送给监听的消息中间件（实时主动推送，rabbitmq需要是在线状态）

2.4 实现步骤

2.4.1 准备材料

1、需要部署一个阿里巴巴的Canal服务端，用于订阅Mysql binlog日志并推送到MQ消息队列

```
#下载解压canal server
wget https://github.com/alibaba/canal/releases/download/canal-1.1.5-alpha-
1/canal.deployer-1.1.5-SNAPSHOT.tar.gz  #下载canal部署包
mkdir canal
tar -zxvf canal.deployer-1.1.1.tar.gz -C canal #解压

#编辑conf/canal.properties，修改MQ配置

canal.ip = 1 #canal服务器标识
canal.serverMode = rabbitmq # 指定rabbitmq
canal.mq.servers = 192.168.223.128 ## 注意不要加端口号，不然会报IPV6错误。
canal.mq.vhost=canal  #MQ虚拟机名称
canal.mq.exchange=exchange.trade #交换机名称，用于将消息发送到绑定的队列
canal.mq.username=guest #MQ登录账号，注意要有上面vhost的权限
canal.mq.password=guest #MQ密码
----------------------------------------------------------------------------------
-

#编辑conf/example/instance.properties实例配置,配置数据库信息
ce.dbUsername=root
canal.instance.dbPassword=root
canal.instance.mysql.slaveId=1234 #不要与my.cnf中server_id相同,因为我要伪装为mysql的
slave
canal.instance.master.address=192.168.223.128:3306 ## 数据库地址
canal.instance.defaultDatabaseName=test ## 数据库名
canal.mq.topic=example # 路由键，需要跟MQ中交换机队列的绑定路由key保持一致
```

2、安装RabbitMQ

```
mkdir /usr/local/rabbitmq;
cd /usr/local/rabbitmq;

yum install build-essential openssl openssl-devel unixODBC unixODBC-devel make
gcc gcc-c++ kernel-devel m4 ncurses-devel tk tc xz tcp_wrappers;

wget www.rabbitmq.com/releases/erlang/erlang-18.3-1.el7.centos.x86_64.rpm;

wget http://repo.iotti.biz/CentOS/7/x86_64/socat-1.7.3....

wget www.rabbitmq.com/releases/rabbitmq-server/v3.6.5/rabbitmq-server-3.6.5-
1.noarch.rpm;

rpm -ivh erlang-18.3-1.el7.centos.x86_64.rpm;

rpm -ivh socat-1.7.3.2-5.el7.lux.x86_64.rpm;

rpm -ivh rabbitmq-server-3.6.5-1.noarch.rpm;

vim /usr/lib/rabbitmq/lib/rabbitmq_server-3.6.5/ebin/rabbit.app
```

找到loopback_users 修改后台登录用户为[guest]

启动rabbitmq：service rabbitmq-service start
启动监控管理器：service rabbitmq-plugins enable rabbitmq_management

开启端口：firewall-cmd --zone=public --add-port=15672/tcp --permanent ；
重启防火墙：firewall-cmd --reload；

3、安装mysql并开启binlog

```
rpm -Uvh http://dev.mysql.com/get/mysql-community-release-el7-5.noarch.rpm #下载
yum -y install mysql-community-server #rpm安装

#编辑my.cnf配置文件，开启binlog
vim /etc/my.cnf

#增加以下配置
log-bin=mysql-bin # 开启 binlog
binlog-format=ROW # 选择 ROW 模式
server_id=1 # 配置 MySQL replaction 需要定义，不要和 canal 的 slaveId 重复（没有数据库主从不配也行）

#加入开机启动
systemctl enable mysqld
#启动mysql服务进程
systemctl start mysqld
#初始化,执行命令,重置密码
mysql_secure_installation
#会依次出现以下问题。
    Set root password? [Y/n]
    是否设置root用户的密码（y后【设置登录密码】）
    Remove anonymous users? [Y/n]
    是否删除匿名用户（y）
    Disallow root login remotely? [Y/n]
    是否禁止root远程登录（n）
    Remove test database and access to it? [Y/n]
    是否删除test数据库(y)
    Reload privilege tables now? [Y/n]
    是否重新加载授权信息（y）

# 先进入mysql
mysql -u root -p
# 授权(root用户)远程连接权限(不建议)
GRANT ALL PRIVILEGES ON *.* TO 'root'@'%' IDENTIFIED BY '远程登录密码' WITH GRANT OPTION;
FLUSH PRIVILEGES;
# 使用单独的远程登录用户(推荐)
GRANT ALL PRIVILEGES ON *.* TO '新用户名'@'%' IDENTIFIED BY '远程登录密码' WITH GRANT OPTION;
FLUSH PRIVILEGES;

#查看是否已经开启了binlog日志
#登录mysql后输入如下命令：
show variables like '%log_bin%';

| Variable_name                  | Value                          |
+--------------------------------+--------------------------------+
```

```
| log_bin                         | ON                              |
| log_bin_basename                | /var/lib/mysql/mysql-bin        |
| log_bin_index                   | /var/lib/mysql/mysql-bin.index |
| log_bin_trust_function_creators | OFF                             |
| log_bin_use_v1_row_events       | OFF                             |
| sql_log_bin                     | ON

#查看binlog日志：
#1、查看第一个binlog文件的内容
mysql> show binlog events;
+------------------+-----+-------------+-----------+-------------+--------------
-------------------------+
| Log_name         | Pos | Event_type  | Server_id | End_log_pos | Info
                        |
+------------------+-----+-------------+-----------+-------------+--------------
-------------------------+
| mysql-bin.000001 |   4 | Format_desc |         1 |         120 | Server ver:
5.6.49-log, Binlog ver: 4 |
| mysql-bin.000001 | 120 | Query       |         1 |         192 | BEGIN
                       |
| mysql-bin.000001 | 192 | Table_map   |         1 |         249 | table_id: 70
(test.goods_store)       |
| mysql-bin.000001 | 249 | Delete_rows |         1 |         294 | table_id: 70
flags: STMT_END_F        |
| mysql-bin.000001 | 294 | Xid         |         1 |         325 | COMMIT /*
xid=11 */                  |
+------------------+-----+-------------+-----------+-------------+--------------
-------------------------+
```

分别启动mysql，rabbitmq，canal

```
service mysql start #启动mysql
service rabbitmq-server start #启动rabbitmq
canal目录/bin/startup.sh #启动canal服务
```

问题：

```
 {"identity":{"slaveId":-1,"sourceAddress":
{"address":"ydt1","port":3306}},"postion":
{"gtid":"","included":false,"journalName":"mysql-
bin.000026","position":551,"serverId":1,"timestamp":1594283137000}}
2020-07-31 17:27:24.973 [destination = example , address = /192.168.223.128:3306
, EventParser] WARN  c.a.o.c.p.inbound.mysql.rds.RdsBinlogEventParserProxy - ---
> find start position successfully,
EntryPosition[included=false,journalName=mysql-
bin.000026,position=551,serverId=1,gtid=,timestamp=1594283137000] cost : 617ms ,
the next step is binlog dump
2020-07-31 17:27:25.106 [destination = example , address = /192.168.223.128:3306
, EventParser] ERROR c.a.o.canal.parse.inbound.mysql.dbsync.DirectLogFetcher -
I/O error while reading from client socket

#如果出现了以上问题，可能是mysql数据库的binlog日志位置不对，重新设置一下
#先找出当前mysql的binlog日志position，进入mysql客户端，输入如下命令：
show master status;
#找到当前binlog以及position，编辑canal目录/conf/example/meta.dat
vim /usr/local/canal/conf/example/meta.dat
将----》"journalName":"mysql-bin.000003","position":499改为自己查到的或者比查到的小即
可
```

2.4.2 代码实现

pom.xml

```
<dependency>
            <groupId>org.mybatis.spring.boot</groupId>
            <artifactId>mybatis-spring-boot-starter</artifactId>
            <version>2.1.2</version>
        </dependency>

        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>8.0.16</version>
        </dependency>

        <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>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-amqp</artifactId>
        </dependency>
```

```
        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>fastjson</artifactId>
            <version>1.2.3</version>
        </dependency>
```

spring配置文件application.yml

```
spring:
  rabbitmq:
    virtual-host: canal
    host: 192.168.223.128
    publisher-confirms: true
  #数据源
  datasource:
    url: jdbc:mysql://192.168.223.128:3306/test?
useUnicode=true&characterEncoding=utf8&useSSL=false&serverTimezone=UTC
    username: root
    password: root
    driver-class-name: com.mysql.jdbc.Driver
  redis:
    host: 192.168.223.128
```

消息队列配置类:

```
package com.ydt.test.message;

import org.springframework.amqp.core.Binding;
import org.springframework.amqp.core.BindingBuilder;
import org.springframework.amqp.core.DirectExchange;
import org.springframework.amqp.core.Queue;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;


@Configuration
public class DirectRabbitConfig {

    //队列 起名：exchange.trade.canal
    @Bean
    public Queue TestDirectQueue() {
        return new Queue("exchange.trade.canal",true);
    }

    //Direct交换机 起名：exchange.trade
    @Bean
    DirectExchange TestDirectExchange() {
        return new DirectExchange("exchange.trade");
    }

    //绑定  将队列和交换机绑定, 并设置用于匹配键：example
    @Bean
    Binding bindingDirect() {
```

```
        return
BindingBuilder.bind(TestDirectQueue()).to(TestDirectExchange()).with("example");
    }
}
```

消息监听者类:

```
package com.ydt.test.message;

import com.alibaba.fastjson.JSON;
import com.alibaba.fastjson.JSONArray;
import com.alibaba.fastjson.JSONObject;
import com.rabbitmq.client.Channel;
import org.springframework.amqp.core.Message;
import org.springframework.amqp.core.MessageProperties;
import org.springframework.amqp.rabbit.annotation.RabbitHandler;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.stereotype.Component;
import org.springframework.util.StringUtils;

import java.io.UnsupportedEncodingException;
import java.util.HashMap;
import java.util.Map;

@Component
public class DirectReceiver {

    @Autowired
    private RedisTemplate redisTemplate;

    //@RabbitListener(queues = "exchange.trade.canal")//监听的队列名称
exchange.trade.canal
    public void process(Message message) throws UnsupportedEncodingException {
        String json  = new String(message.getBody());
        System.out.println("DirectReceiver消费者收到消息  : " + json);
        Map map = JSON.parseObject(json,Map.class);
        JSONArray array = null;
        String sqlType = (String) map.get("type");
        if(StringUtils.endsWithIgnoreCase("SELECT",sqlType)){
            array = JSONArray.parseArray((String)map.get("data"));
        }else{
            array = (JSONArray)map.get("data");
        }
        if(array == null){
            return;
        }
        JSONObject object = array.getJSONObject(0);
        if(StringUtils.endsWithIgnoreCase("UPDATE",sqlType)
                || StringUtils.endsWithIgnoreCase("INSERT",sqlType)
                    || StringUtils.endsWithIgnoreCase("SELECT",sqlType)){

 redisTemplate.boundValueOps(object.get("code")).set(object.toString());
        }else if(StringUtils.endsWithIgnoreCase("DELETE",sqlType)){
            redisTemplate.delete(object.get("code"));
        }
```

```
    }
}
```

Controller来两个方法：

```
package com.ydt.test.controller;

import com.alibaba.fastjson.JSON;
import com.ydt.test.domain.Store;
import com.ydt.test.mapper.StoreMapper;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.util.StringUtils;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.HashMap;
import java.util.Map;

@RestController
public class MessageController {


    @Autowired
    private StoreMapper storeMapper;

    @Autowired
    private RabbitTemplate rabbitTemplate;

    @Autowired
    private RedisTemplate redisTemplate;

    /**
     * 案例1
     */
    @RequestMapping("update1")
    public void update1(String code,int store){
        Store storeObj =  new Store(code,store);
        storeMapper.update(storeObj);
        int i = 1/0;

 redisTemplate.opsForValue().set(storeObj.getCode(),JSON.toJSONString(store));

    }

    /**
     * 案例2 延迟双删
     */
    @RequestMapping("update2")
    public void update2(String code,int store){
        redisTemplate.delete(code);//先将缓存数据清空
        Store oldStore = storeMapper.getStore(code);
        Store storeObj =  new Store(code,store);
        storeMapper.update(storeObj);//更新数据
```

```
        try {
            Thread.sleep(3000);//这个地方需要自己去评估项目的读数据业务逻辑的耗时，然后加几百ms
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

 redisTemplate.opsForValue().append(oldStore.getCode(),JSON.toJSONString(oldStore));//模拟其他线程弄进去的脏数据
        System.out.println("延迟删除");
        Boolean isdelete = false;
        while (!isdelete){//重试机制
            isdelete = redisTemplate.delete(code);//再次将数据清除
        }
        System.out.println("缓存删除成功");
    }


    @RequestMapping("get")
    public String getMessage(){
        //查询操作
        Store store = storeMapper.getStore("20200101010101");
        System.out.println("-----------我进行了查询，现在我要开始进行redis缓存了----------");
        //同一数据源
        Map map = new HashMap();
        map.put("type", "SELECT");
        map.put("data", "[{'code':'20200101010101','store':"+store.getStore()+"}]");
        rabbitTemplate.convertAndSend("exchange.trade", "example", JSON.toJSONString(map));
        return  "";
    }
}
```

启动测试

1、先将库存表中库存修改为111，会通过canal伪slave拿到binlog日志，然后推送到rabbitmq

2、然后调用get方法，拿到数据库中数据，同时将数据推送到rabbitmq

3、重复1操作，将库存改为321

4、重复2操作

打开监听，启动服务可以看到我们的消费者会按照顺序消费队列中的数据！

```
DirectReceiver消费者收到消息  : {"data":
[{"code":"20200101010101","store":"111"}],"database":"test","es":1596198062000,"
id":13,"isDdl":false,"mysqlType":
{"code":"varchar(255)","store":"int(11)"},"old":[{"store":"123"}],"pkNames":
["code"],"sql":"","sqlType":
{"code":12,"store":4},"table":"goods_store","ts":1596198062527,"type":"UPDATE"}
DirectReceiver消费者收到消息  : {"data":"
[{'code':'20200101010101','store':111}]","type":"SELECT"}
DirectReceiver消费者收到消息  : {"data":
[{"code":"20200101010101","store":"321"}],"database":"test","es":1596198075000,"
id":14,"isDdl":false,"mysqlType":
{"code":"varchar(255)","store":"int(11)"},"old":[{"store":"111"}],"pkNames":
["code"],"sql":"","sqlType":
{"code":12,"store":4},"table":"goods_store","ts":1596198075520,"type":"UPDATE"}
DirectReceiver消费者收到消息  : {"data":"
[{'code':'20200101010101','store':321}]","type":"SELECT"}
```