

# 项目介绍 — 电商实时分析系统

## 项目背景

本项目主要用于互联网电商企业中，使用 Flink 技术开发的大数据统计分析平台，对电商网站的各种用户行为（访问行为、购物行为、点击行为等）进行复杂的分析，用统计分析出来的数据，辅助公司中的 PM（产品经理）、数据分析师以及管理人员分析现有产品的情况，并根据用户行为分析结果持续改进产品的设计，以及调整公司的战略和业务。最终达到用大数据技术来帮助提升公司的业绩、营业额以及市场占有率的目标。

## 项目架构



## 项目的技术选型

为什么要选择架构中的技术？

Kafka

Hbase

Canal

Flink

Kafka

- 吞吐量高

  - 每秒达到几十万的数据

- 速度快

  - 内存接力（消费者直接消费 PageCache 内存区数据）

  - 顺序刷入磁盘

- 数据安全

  - 冗余机制可以有效避免丢失错误

  - 其中几台实例挂掉也可以继续工作

- 高并发

  - 突发型流量可以将峰值的数据持久化，等到流量正常或低估时再取出消费

Hbase

- 业务不复杂

  - PV/UV、页面点击量、新鲜度（没有复杂 SQL 查询）

- 实时电商平台

  - 存取海量的数据

- 社区活跃

  - Hbase 社区非常大，Facebook、小米、网易等公司都在使用 HBase

- 高可用

  - 没有单点故障，高可用性高

Canal

数据库同步常用的有两种方案：

方案 1

mysql --> logstash --> kafka --> flink --> hbase

方案 2

mysql --> sqoop --> hbase

### 上述方案存在的问题

logstash、sqoop 还是需要 使用 SQL 语句查询 mysql ，会给 mysql 增加压力，如果要跑大量数据的同步，会拖垮 mysql  
解决方案

mysql --> cannal(binlog) --> kafka --> flink --> hbase

cannal

Canal 可以实时解析 mysql 的 binlog 日志，通过读取 binlog 日志，将数据输出到 Kafka。 不需要执行 SQL 语句 ，不会增加 mysql 压力

Flink

- 速度要比 Spark、MapReduce 更快

- 保证 EXACTLY\_ONCE

容错更轻量级

自动程序调优，自动避免性能消耗较大的操作（例如：shuffle、sort）

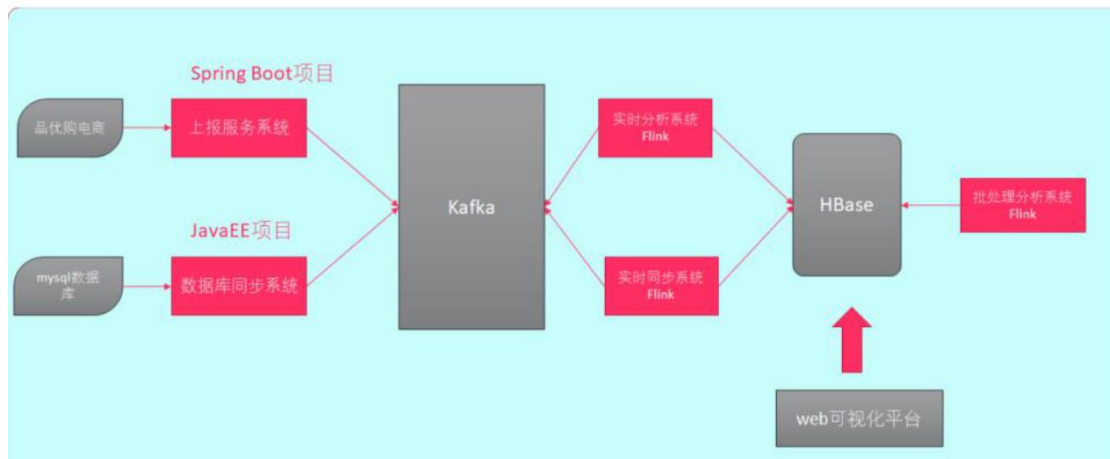
高吞吐

## 课程目标

- 1)：掌握 HBASE 的搭建和基本运维操作
- 2)：掌握 flink 基本语法
- 3)：掌握 kafka 的搭建和基本运维操作
- 4)：掌握 canal 的使用
- 5)：能够独立开发出上报服务
- 6)：能够使用flink：处理实时热点数据及数据落地 Hbase
- 7)：能够使用flink：处理频道的 PV、UV 及数据落地 Hbase
- 8)：能够使用flink：处理新鲜度
- 9)：能够使用flink：处理频道地域分布
- 10)：能够使用flink：处理运营商平台数据
- 11)：能够使用flink：处理浏览器类型
- 12)：能够使用代码对接 canal，并将数据同步到 kafka
- 13)：能够使用flink 同步数据到 hbase

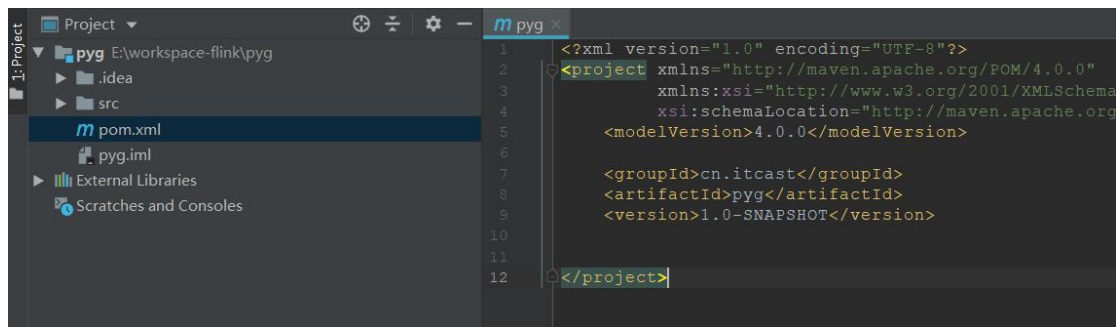
## 项目整体工程搭建

### 工程结构

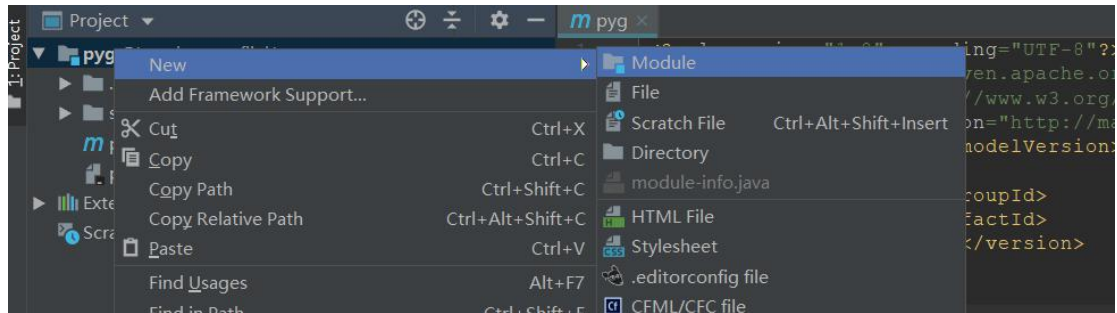


### 工程创建

- 2.1: 创建一个 maven 工程：pyg

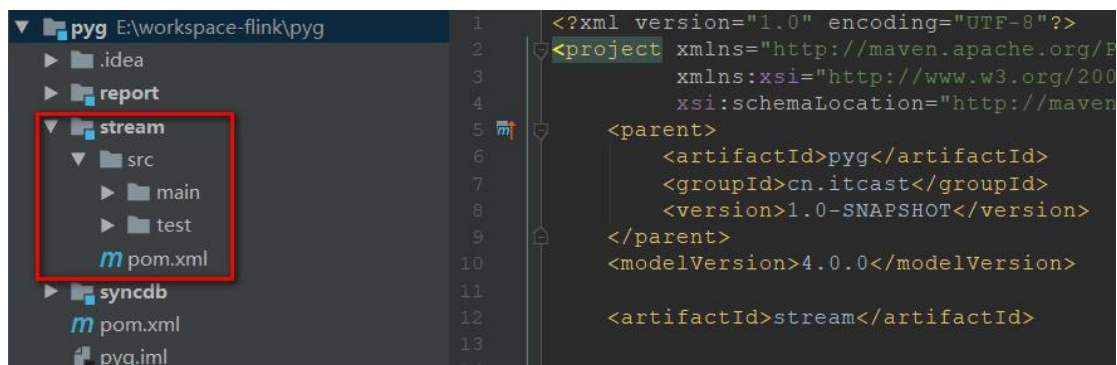


## 2.2: 创建 maven子工程：选择 Module



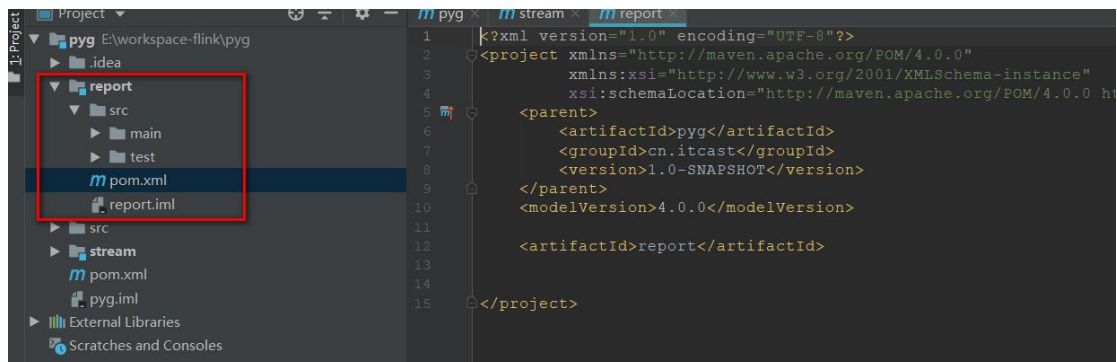
创建子工程：实时流处理业务模块，项目名：stream

选择 Module -->maven



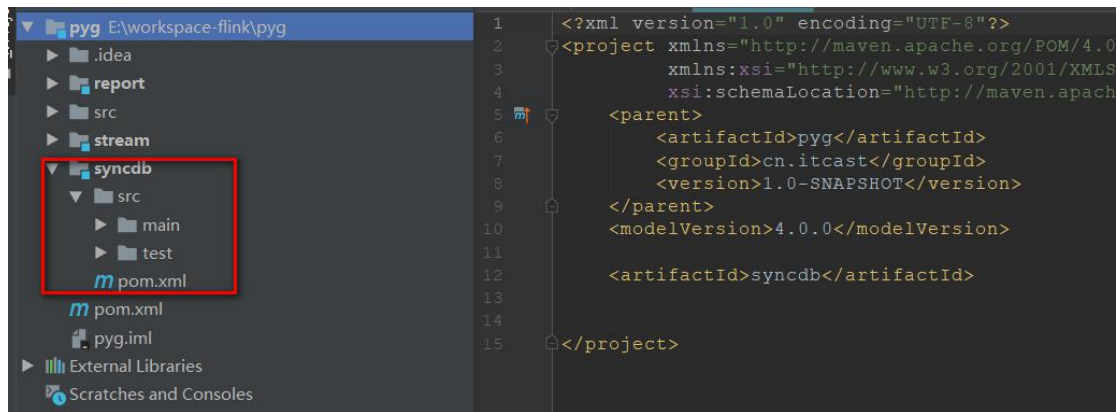
## 2.2: 创建子工程：上报服务模块(report)

选择 Module -->maven



## 2.2: 创建子工程：数据库同步模块(syncdb)

选择 Module -->maven



## 上报服务系统开发

### Spring Boot 简介

Spring Boot 是一个基于 Spring 之上的快速应用构建框架。使用 Spring Boot 可以快速开发出基于 Spring 的应用。

Spring Boot 主要解决两方面的问题：

依赖太多问题

轻量级 JavaEE 开发，需要导入大量的依赖

依赖之间还存在版本冲突

配置太多问题

大量的 XML 配置

### 开发 Spring Boot 程序的基本步骤

导入 Spring Boot 依赖

编写 `application.properties` 配置文件

编写 `Application` 入口程序

### 配置 Maven 本地仓库

因为网络下载慢的原因，所以 建议 使用提供给大家的本地仓库，直接使用来开发项目

1. 将 资料 中的 `maven_本地仓库.zip` 解压
2. 并在 Maven、IDEA 中配置该 本地仓库

### 导入 Maven 依赖

从 资料\文件\上报服务 目录中的 `pom.xml` 拷贝依赖到 上报服务系统项目 中

## 编写 Spring Boot 启动类

编写 `Report`，作为用来启动 `springBoot` 的入口

```
package cn.itcast;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

/**
 * @Date 2019/5/29
 */
@SpringBootApplication
public class Report {

    public static void main(String[] args) {
        SpringApplication.run(Report.class, args);
    }

}
```

编写 `application.properties`，制定应用名称和端口

```
server.port=6097
spring.application.name=ReportApplication
```

## 验证 Spring Boot 是否创建成功

编写 `Test`，用来收发数据

```
package cn.itcast;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

/**
 * @Date 2019/5/30
 */
@Controller
@RequestMapping("report")
public class Test {

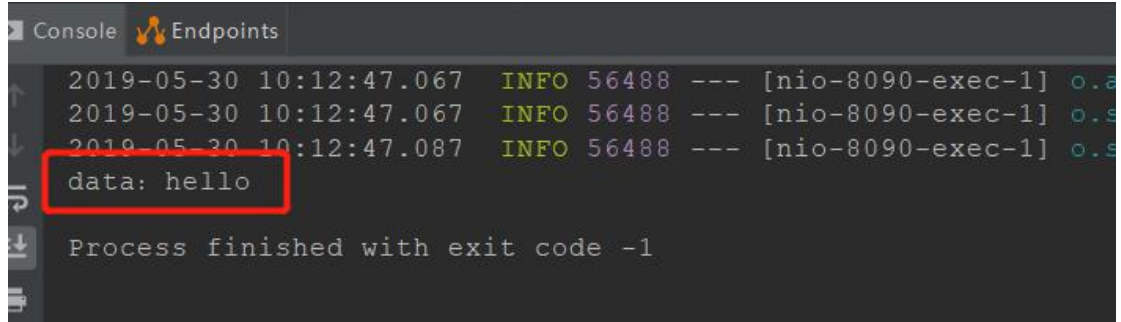
    @RequestMapping("test")
    public void receiveData(String data) {
        System.out.println("data: " + data);
    }

}
```

```
}  
}
```

运行APP，然后在页面输入：<http://localhost:8090/report/test?data=hello>

控制台打印如下日志，说明项目搭建成功：



```
Console  Endpoints  
2019-05-30 10:12:47.067 INFO 56488 --- [nio-8090-exec-1] o.a...  
2019-05-30 10:12:47.067 INFO 56488 --- [nio-8090-exec-1] o.s...  
2019-05-30 10:12:47.087 INFO 56488 --- [nio-8090-exec-1] o.s...  
data: hello  
Process finished with exit code -1
```

## springBoot 整合 kafka

在 application.properties 中添加配置文件：

```
server.port=8090  
spring.application.name=report  
#===== kafka producer=====  
#kafka 的服务器地址  
kafka.producer.servers=node01:9092,node02:9092,node03:9092  
#如果出现发送失败的情况，允许重试的次数  
kafka.producer.retries=0  
#每个批次发送多大的数据  
kafka.producer.batch.size=4096  
#定时发送，达到 1ms 发送  
kafka.producer.linger=1  
#缓存的大小  
kafka.producer.buffer.memory=40960
```

## 编写 KafkaProducerConfig

```
package com.pyg.report.Controller;  
  
import org.apache.kafka.clients.producer.ProducerConfig;  
import org.apache.kafka.common.serialization.StringSerializer;  
import org.springframework.beans.factory.annotation.Value;  
import org.springframework.context.annotation.Bean;
```

```

import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.annotation.EnableKafka;
import org.springframework.kafka.core.DefaultKafkaProducerFactory;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.kafka.core.ProducerFactory;
import java.util.HashMap;
import java.util.Map;

@Configuration
@EnableKafka

public class KafkaProducerConfig {

    @Value("${kafka.producer.servers}")
    private String servers;

    @Value("${kafka.producer.retries}")
    private int retries;

    @Value("${kafka.producer.batch.size}")
    private int batchSize;

    @Value("${kafka.producer.linger}")
    private int linger;

    @Value("${kafka.producer.buffer.memory}")
    private int bufferMemory;

    public Map<String, Object> producerConfigs() {
        Map<String, Object> props = new HashMap<>();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, servers);
        props.put(ProducerConfig.RETRIES_CONFIG, retries);
        props.put(ProducerConfig.BATCH_SIZE_CONFIG, batchSize);
        props.put(ProducerConfig.LINGER_MS_CONFIG, linger);
        props.put(ProducerConfig.BUFFER_MEMORY_CONFIG, bufferMemory);
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
            StringSerializer.class);
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
            StringSerializer.class);
        return props;
    }

    public ProducerFactory<String, String> producerFactory() {
        return new DefaultKafkaProducerFactory<String, String>(producerConfigs());
    }

    @Bean

    public KafkaTemplate<String, String> kafkaTemplate() {
        return new KafkaTemplate<String, String>(producerFactory());
    }
}

```



## 编写上报服务

### 添加 Message

```
package com.pyg.report.msg;

/**
 * Created by angel
 */
public class Message {
    private String message;//json 格式的消息内容
    private int count;//消息的次数
    private Long timestamp;//消息的时间
    public String getMessage() {
        return message;
    }
    public void setMessage(String message) {
        this.message = message;
    }
    public int getCount() {
        return count;
    }
    public void setCount(int count) {
        this.count = count;
    }
    public Long getTimestamp() {
        return timestamp;
    }
    public void setTimestamp(Long timestamp) {
        this.timestamp = timestamp;
    }
    @Override
    public String toString() {
        return "Message{" +
            "message='" + message + '\'' +
            ", count=" + count +
            ", timestamp=" + timestamp +
            '}';
    }
}
```

## 编写 ReportApplication

```
package cn.itcast.Controller;

import com.alibaba.fastjson.JSON;
import com.pyg.report.msg.Message;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.http.HttpStatus;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.OutputStream;
import java.io.PrintWriter;
import java.util.Date;

@SpringBootApplication
@Controller
@RequestMapping("/report")
public class ReportApplication {

    @Autowired

    private KafkaTemplate kafkaTemplate;

    /**
     * @param json 接收的数据
     * @param request
     * @param response
     */
    @RequestMapping(value = "put", method = RequestMethod.POST)
    public void retrieveData(@RequestBody String json, HttpServletResponse response) {

        Message msg = new Message();
        msg.setMessage(json);
        msg.setCount(1);
        msg.setTimestamp(new Date().getTime());
        json = JSON.toJSONString(msg);
        System.out.println(json);

        //业务开始
        kafkaTemplate.send("test", "key", json);

        //业务结束

        PrintWriter printWriter = getWriter(response);
        response.setStatus(HttpStatus.OK.value());
        printWriter.write("success");
    }
}
```

```

        close(printWriter);
    }

    private PrintWriter getWriter(HttpServletResponse response) {
        response.setCharacterEncoding("utf-8");
        response.setContentType("application/json");
        OutputStream out = null;
        PrintWriter printWriter = null;
        try {
            out = response.getOutputStream();
            printWriter = new PrintWriter(out);
        } catch (IOException e) {
            e.printStackTrace();
        }
        return printWriter;
    }

    private void close(PrintWriter printWriter) {
        printWriter.flush();
        printWriter.close();
    }
}

```

## 添加 HttpTest

```

package cn.itcast.test;

import java.io.BufferedOutputStream;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.HttpURLConnection;
import java.net.URL;

/**
 *编写测试类:
 * Created by angel
 */
public class HttpTest{

    public static void main(String[] args) {
        String url = "http://localhost:8090/report/put";
        sendData(url, "send success");
    }

    public static void send(String address,String message) {
        try {
            URL url = new URL(address);

```

```

        HttpURLConnection conn = (HttpURLConnection)url.openConnection();
        conn.setRequestMethod("POST");
        conn.setDoInput(true);
        conn.setDoOutput(true);
        conn.setAllowUserInteraction(true);
        conn.setUseCaches(false);
        conn.setReadTimeout(6*1000);
        conn.setRequestProperty("User-Agent", "Mozilla/5.0 (Windows NT
        6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/68.0.3440.106
        Safari/537.36");
        conn.setRequestProperty("Content-Type", "application/json");
        conn.connect();
        OutputStream outputStream = conn.getOutputStream();
        BufferedOutputStream out = new BufferedOutputStream(outputStream);
        out.write(message.getBytes());
        out.flush();
        String temp = "";
        InputStream in = conn.getInputStream();
        byte[] tempbytes = new byte[1024];
        while(in.read(tempbytes, 0, 1024) != -1){
            temp+=new String(tempbytes);
        }
        System.out.println(temp);
        System.out.println("<<<响应码: "+conn.getResponseCode());
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}

```

测试类打印:

```

<<<响应码: 200

Process finished with exit code 0

```

说明整合成功!

## 安装 Kafka-Manager

Kafka-manager 是 Yahoo!开源的一款 Kafka 监控管理工具。

### 步骤

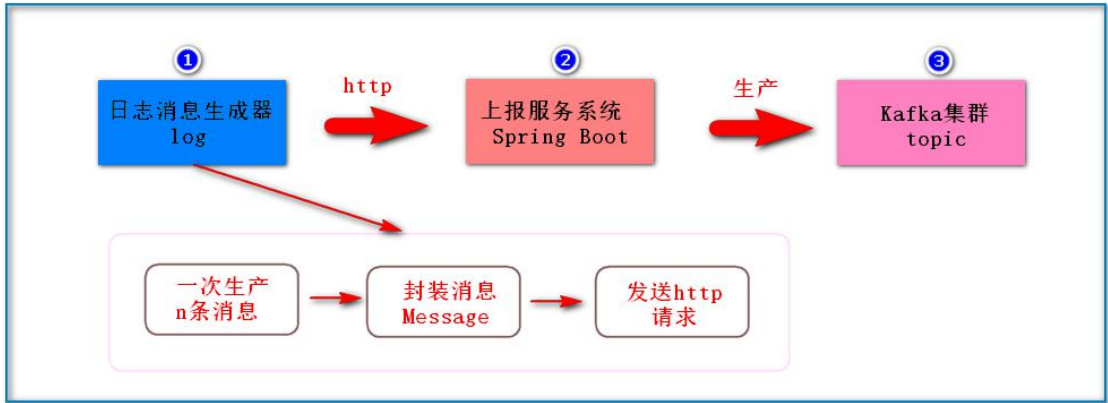
1. 上传 [资料\软件包](#) 中的 [kafka-manager-1.3.3.23.zip](#)

- 2. 解压到 /export/servers
- 3. 修改 conf/application.conf  
kafka-manager.zkhosts="node01:2181,node02:2181,node03:2181"
- 4. 启动 zookeeper
- 5. 启动 kafka
- 6. 直接运行 bin/kafka-manager
- 7. 浏览器中使用 node01:9000 访问即可

默认 kafka-manager 的端口号为 9000，如果该端口被占用，请使用下面的命令修改端口 bin/kafka-manager -Dconfig.file=/export/servers/kafka-manager-1.3.3.23/conf/application.conf -Dhttp.port=10086

## 模拟生产点击流日志消息到 Kafka

为了方便进行测试，需要有一个消息生成工具来生成点击流日志，然后发送给上报服务系统。



### 步骤

- 1. 导入 day03\资料\文件\上报服务 中的点击实体类 (UserBrowse.java)
- 2. 导入 day03\资料\文件\上报服务 中的点击流日志生成器 (UserBrowseRecord.java)

### 点击流日志字段

字段	说明
channelID	频道 ID
categoryID	产品的类别 ID
produceID	产品 ID
country	国家
province	省份
city	城市
network	网络方式 (移动, 联通, 电信...)
source	来源方式
browserType	浏览器类型
entryTime	进入网站时间

leaveTime	离开网站时间
userID	用户 ID

验证测试代码

步骤

- 1. 创建 Kafka 的 topic ( report )
- 2. 启动 ZooKeeper 集群
- 3. 启动 Kafka 集群
- 4. 使用 kafka-console-consumer.sh 消费 topic 中的数据
- 5. 重新启动上报服务
- 6. 执行 UserBrowseRecord.java 的 main 方法，生成一百条用户浏览消息到 Kafka

创建 kafka topic

```
bin/kafka-topics.sh --create --zookeeper node01:2181 --replication-factor 2 --partitions 3 --topic report
```

启动 kafka 消费者

```
bin/kafka-console-consumer.sh --zookeeper node01:2181 --from-beginning --topic report
```

kafka 避免数据倾斜

```
public class RoundRobinPartition implements Partitioner {
    // 计数器，每次产生一条消息+1
    AtomicInteger atomicInteger = new AtomicInteger(0);

    public int partition(String topic, Object key, byte[] keyBytes, Object value, byte[] valueBytes, Cluster cluster) {
        // 获取分区数量
        Integer partitionCount = cluster.partitionCountForTopic(topic);
        int partitions = atomicInteger.incrementAndGet() % partitionCount;

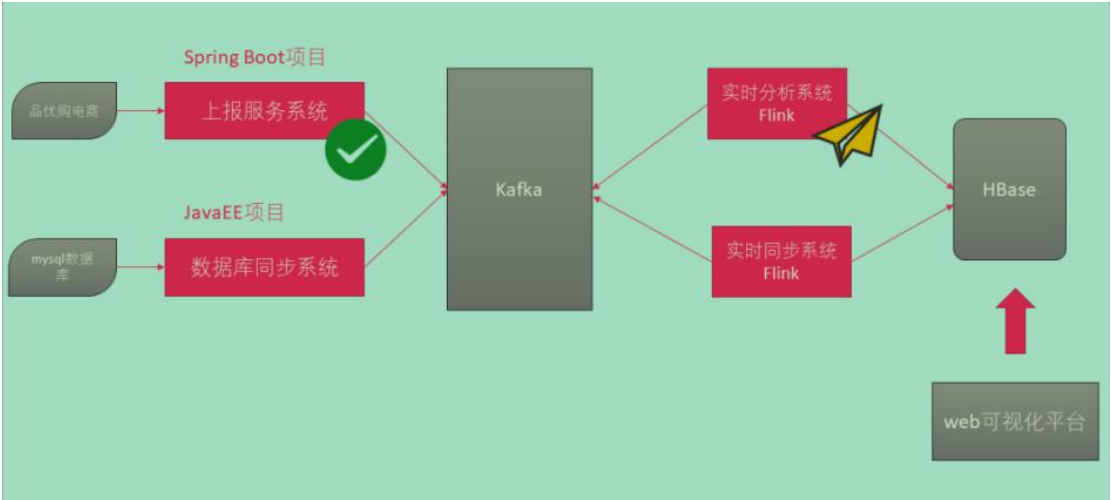
        if( atomicInteger.get()>2000){
            atomicInteger.set(0);
        }

        return partitions;
    }

    public void close() {}

    public void configure(Map<String, ?> configs) {}
}
```

# Flink 实时数据处理系统开发



## 业务

- 实时分析频道热点
- 实时分析频道 PV/UV
- 实时分析频道新鲜度
- 实时分析频道地域分布
- 实时分析运营商平台
- 实时分析浏览器类型

## 技术

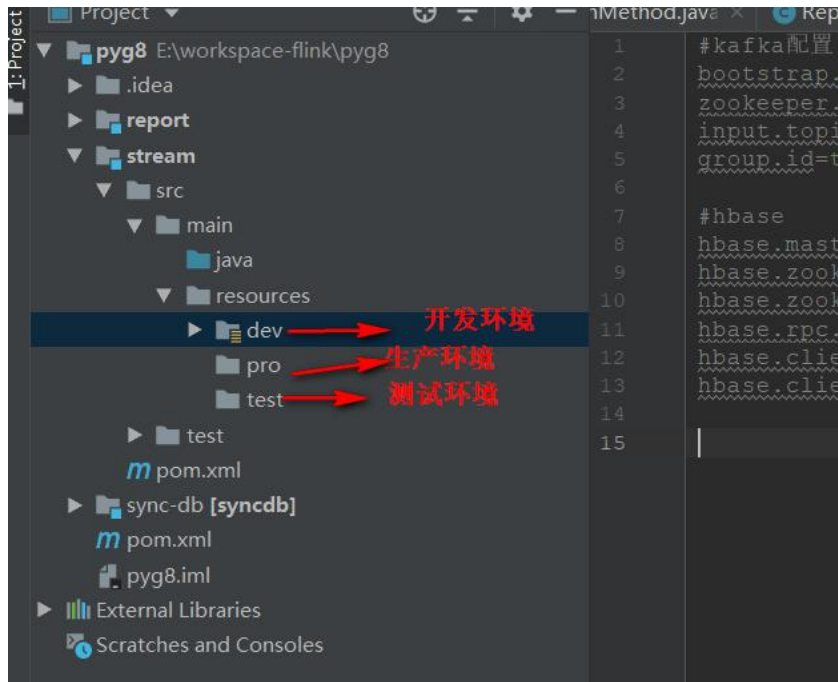
- Flink 实时处理算子
- 使用 CheckPoint 和 水印 解决 Flink 生产上遇到的问题（网络延迟、丢数据）
- Flink 对接 Kafka
- Flink 对接 HBase

# 搭建 Flink 实时数据处理系统项目环境

## 导入 Maven 项目依赖

1. 将 `day03\资料\文件\实时流处理` 目录的 `pom.xml` 文件中的依赖导入到 `stream` 项目的 `pom.xml`
2. `stream` 模块添加 `scala` 支持
3. `main` 创建 `scala` 文件夹，并标记为源代码目录

环境介绍：



## 创建项目包结构

包名	说明
com.itcast.util	存放工具类、配置工具（读取 *.properties）类
com.itcast.bean	存放实体类
com.itcast.task	存放具体的分析任务 每一个业务都是一个任务，对应的分析处理都写在这里
com.itcast.map	流数据转换成对象
com.itcast.reduce	存放数据聚合对象
com.itcast.sink	数据落地、存储对象
com.itcast.trit	数据统一接口

### 导入配置文件

1. 将 day03\资料\文件\实时流处理 目录中的 application.properties 导入到 resources 目录
2. 将 day03\资料\文件\实时流处理 目录中的 log4j.properties 到 resources 目录下

注意修改 kafka 服务器 和 hbase 服务器 的机器名称



编写配置类

获取配置文件 API 介绍

ConfigFactory.load() 介绍

- 1、使用 ConfigFactory.load() 可以自动加载配置文件中的 application.properties 文件，并返回一个 Config 对象，
- 2、使用 Config 对象可以获取到配置文件中的配置项
- 3、application.properties 文件存放 key-value 键值对的数据

在 com.itcast.config 包下创建 GlobalConfigUtil 单例对象 (object)

步骤

- 1. 使用 ConfigFactory.load 获取配置对象
- 2. 编写方法加载 application.properties 配置
- 3. 添加一个 main 方法测试 工具类是否能够正确读取配置项

API

方法名	说明
getString("key")	获取配置文件中指定 key 的值对应的字符串
getInt("key")	获取配置文件中指定 key 的值对应的整型数字
getLong("key")	同上
getBoolean("key")	同上

示例代码：

```
package cn.itcast.config

import com.typesafe.config.{Config, ConfigFactory}

/**
 * @Date 2019/5/29
 */
object GlobalConfig {
    private val config: Config = ConfigFactory.load()

    //kafka
    val kServer: String = config.getString("bootstrap.servers")
    val kZk: String = config.getString("zookeeper.connect")
    val kTopic: String = config.getString("input.topic")
    val kGroupID: String = config.getString("group.id")

    //Hbase
    val hmaster: String = config.getString("hbase.master")
    val hZk: String = config.getString("hbase.zookeeper.quorum")
    val hPoint: String = config.getString("hbase.zookeeper.property.clientPort")
}
```

```
val hRpc: String = config.getString("hbase.rpc.timeout")

val hClientTimeout: String = config.getString("hbase.client.operation.timeout")

val hScanTimeout: String = config.getString("hbase.client.scanner.timeout.period")

}
```

## APP 驱动类开发

### 步骤

1. 创建 `App` 单例对象，初始化 Flink 运行环境
2. 创建 `main` 方法，获取 `StreamExecutionEnvironment` 运行环境
3. 设置流处理的时间为 `EventTime`，使用数据发生的时间来进行数据处理
4. 将 Flink 默认的开发环境并行度设置为 1
5. 编写测试代码，测试 Flink 程序是否能够正确执行

注意：

1. 一定要导入 `import org.apache.flink.api.scala._` 隐式转换，否则 Flink 程序无法执行
2. 导入 `org.apache.flink.streaming.api` 下的 `TimeCharacteristic`，否则 没有 `EventTime`

## Flink 添加 checkpoint 容错支持

### 步骤

1. Flink environment 中添加 `checkpoint` 支持
2. 运行 Flink 程序测试 checkpoint 是否配置成功（检查 HDFS 中是否已经保存 snapshot 数据）

实现

1. 在 Flink 流式处理环境中，添加以下 checkpoint 的支持，确保 Flink 的高容错性，数据不丢失。

```
// 保证程序长时间运行的安全性进行 checkpoint 操作
//
// 5 秒启动一次 checkpoint
env.enableCheckpointing(5000)
// 设置 checkpoint 强一致性
env.getCheckpointConfig.setCheckpointingMode(CheckpointingMode.EXACTLY_ONCE)
// 设置两次 checkpoint 的最小时间间隔
env.getCheckpointConfig.setMinPauseBetweenCheckpoints(5000)
// checkpoint 超时的时长
env.getCheckpointConfig.setCheckpointTimeout(60000)
// 允许的最大 checkpoint 并行度
env.getCheckpointConfig.setMaxConcurrentCheckpoints(1)
// 当程序关闭的时，触发额外的 checkpoint
env.getCheckpointConfig.enableExternalizedCheckpoints(CheckpointConfig.ExternalizedCheckpointCleanup.RETAIN_ON_CANCELLATION)
// 设置 checkpoint 的地址
```

```
env.setStateBackend(new FsStateBackend("hdfs://node01:8020/checkpoint/"))
```

2. 启动 HDFS
3. 启动 Flink 程序 测试
4. 如果测试成功，在 HDFS 中应该生成如下几个目录

## Browse Directory

/checkpoint/2307a20c27c0ea542d914ee49440957c							Go!
Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
drwxr-xr-x	zbb09	supergroup	0 B	Mon Jun 03 01:35:01 +0800 2019	0	0 B	shared
drwxr-xr-x	zbb09	supergroup	0 B	Mon Jun 03 01:35:01 +0800 2019	0	0 B	taskowned

注意：

1. 修改 HDFS 的 NameNode 节点服务器名称
2. 检查 HDFS 的 NameNode 端口号，并修改 8020
3. MAC 系统：可以执行 `hadoop fs -chmod 777 /`，将所有目录文件权限设置为 777

## Flink 整合 Kafka

### 步骤

1. 配置 Kafka 连接属性
2. 使用 FlinkKafkaConsumer09 整合 Kafka
3. 添加一个 source 到当前 Flink 环境
4. 启动 zookeeper
5. 启动 kafka
6. 运行 Flink 程序测试是否能够从 Kafka 中消费到数据

### 实现

1. 配置 Kafka 连接属性

```
// 整合 Kafka
val properties = new Properties()
properties.setProperty("bootstrap.servers", GlobalConfigUtil.bootstrapServers)
properties.setProperty("zookeeper.connect", GlobalConfigUtil.zookeeperConnect)
properties.setProperty("group.id", GlobalConfigUtil.groupId)
properties.setProperty("enable.auto.commit", GlobalConfigUtil.enableAutoCommit)
properties.setProperty("auto.commit.interval.ms", GlobalConfigUtil.autoCommitIntervalMs)

// 配置下次重新消费的话，从哪里开始消费
// latest: 从上一次提交的 offset 位置开始的
// earliest: 从头开始进行（重复消费数据）
properties.setProperty("auto.offset.reset", GlobalConfigUtil.autoOffsetReset)

// 配置序列化和反序列化
properties.setProperty("key.serializer", "org.apache.kafka.common.serialization.StringSerializer")
properties.setProperty("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer")
```

```
val consumer: FlinkKafkaConsumer09[String] = new FlinkKafkaConsumer09[String](
    GlobalConfigUtil.inputTopic,
    new SimpleStringSchema(),
    properties
)
```

2. 添加一个 source 到当前 Flink 环境

```
val source: DataStream[String] = env.addSource(consumer)
```

3. 打印 DataStream 中的数据

4. 启动 `zookeeper`

5. 启动 `kafka`

6. 运行 Flink 程序

7. 运行上报服务系统，测试是否能够从 Kafka 中消费到数据

如果 Flink 从 Kafka 消费成功会打印以下数据：

```
{ "count": 1, "message": "{ \"browserType\": \"谷歌浏览器\", \"categoryID\": 6, \"channelID\": 4, \"city\": \"America\", \"country\": \"china\", \"entryTime\": 1544601660000, \"leaveTime\": 1544634060000, \"network\": \"联通\", \"produceID\": 4, \"province\": \"china\", \"source\": \"百度跳转\", \"userID\": 13 }\", \"timestamp\": 1553188417573 }
```

## Kafka 消息解析、封装成 Info 对象数据

### 步骤

使用 map 算子，将 kafka 中消费到的数据

使用 FastJSON 转换为 JSON 对象

将 JSON 的数据解析成一个 **Info**

测试是否能够正确解析

### 代码

1. 使用 map 算子，将 kafka 中消费到的数据，使用 FastJSON 转换为 JSON 对象
2. 将 JSON 的数据解析成一个 Info 对象数据

```
//数据转换
val infos: DataStream[Info] = kafkaSource.map(line => {
    val jsonObject = JSON.parseObject(line)
    val message = jsonObject.get("message").toString
    val count = jsonObject.get("count").toString.toInt
    val timestamp: Long = jsonObject.get("timestamp").toString.toLong
    val userCan: UserCan = UserCan.toUserCan(message)
    Info(userCan, timestamp, count)
})
```

3. 测试是否能够正确解析

```
infos.print()
```

## Flink 添加水印处理

1. 在 `App.scala` 中添加水印支持

```
//设置水位线

val waterData: DataStream[Info] = infos.assignTimestampsAndWatermarks(new AssignerWithPeriodicWatermarks[Info] {

    val delayTimestamp = 2000

    var currentTimestamp = 0L

    override def getCurrentWatermark: Watermark = {

        new Watermark(currentTimestamp - delayTimestamp)

    }

    override def extractTimestamp(element: Info, previousElementTimestamp: Long): Long = {

        val timestamp = element.timestamp

        currentTimestamp = Math.max(timestamp, currentTimestamp)

        currentTimestamp

    }

})

waterData.print()
```

2. 启动执行测试

## APP 完整代码：

```
package cn.itcast

import java.util.Properties

import cn.itcast.bean.{Info, UserCan}
import cn.itcast.config.GlobalConfig
import cn.itcast.task.{ChannelHotTask, ChannelPVUVTask}
import com.alibaba.fastjson.JSON
import org.apache.flink.api.common.serialization.SimpleStringSchema
import org.apache.flink.runtime.state.filesystem.FsStateBackend
import org.apache.flink.streaming.api.CheckpointingMode
import org.apache.flink.streaming.api.functions.AssignerWithPeriodicWatermarks
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer09
import org.apache.flink.streaming.api.scala._
import org.apache.flink.streaming.api.watermark.Watermark

/**
 * @Date 2019/5/29
 */
```

```

*/
object App {

  def main(args: Array[String]): Unit = {

    /**
     * 1. 获取执行环境
     * 2. 设置检查点
     * 3. 配置 kafka
     * 4. 数据转换
     * 5. 设置水位线
     * 6. 业务逻辑开发
     * 7. 触发执行
     */

    val env = StreamExecutionEnvironment.getExecutionEnvironment

    env.setStateBackend(new FsStateBackend("hdfs://node01:8020/flink/checkpoint"))
    env.enableCheckpointing(1000)

    env.getCheckpointConfig.setCheckpointingMode(CheckpointingMode.EXACTLY_ONCE)
    env.getCheckpointConfig.setCheckpointInterval(5000)
    env.getCheckpointConfig.setCheckpointTimeout(5000)
    env.getCheckpointConfig.setFailOnCheckpointingErrors(false)
    env.getCheckpointConfig.setMaxConcurrentCheckpoints(1)
    env.getCheckpointConfig.setMinPauseBetweenCheckpoints(5000)

    //配置 kafka

    val properties = new Properties()
    properties.setProperty("bootstrap.servers", GlobalConfig.kServer)
    properties.setProperty("zookeeper.connect", GlobalConfig.kZk)
    properties.setProperty("group.id", GlobalConfig.kGroupID)

    val source = new FlinkKafkaConsumer09[String](GlobalConfig.kTopic, new SimpleStringSchema(), properties)
    val kafkaSource = env.addSource(source)

    //数据转换
    val infos: DataStream[Info] = kafkaSource.map(line => {
      val jsonObject = JSON.parseObject(line)
      val message = jsonObject.get("message").toString
      val count = jsonObject.get("count").toString.toInt
      val timestamp: Long = jsonObject.get("timestamp").toString.toLong
      val userCan: UserCan = UserCan.toUserCan(message)
      Info(userCan, timestamp, count)
    })

    //设置水位线
    val waterData: DataStream[Info] = infos.assignTimestampsAndWatermarks(new AssignerWithPeriodicWatermarks[Info] {

```

```
val delayTimestamp = 2000

var currentTimestamp = 0L

override def getCurrentWatermark: Watermark = {
    new Watermark(currentTimestamp - delayTimestamp)
}

override def extractTimestamp(element: Info, previousElementTimestamp: Long): Long = {
    val timestamp = element.timestamp
    currentTimestamp = Math.max(timestamp, currentTimestamp)
    currentTimestamp
}

})

/**
 * 业务逻辑开发
 * 1. 热点统计
 * 2. pvuv
 * 3. 用户新鲜度统计
 * 4. 区域热点统计
 * 5. 运营商统计
 * 6. 浏览器类型统计
 */
//触发执行
env.execute("App")
}
}
```

## 工具类开发

### HbaseUtil 开发

#### 步骤

(1) 初始化连接

1. 将 `application.properties` 配置文件 copy 到 `resources` 目录
2. 在 `util` 包中添加 `HBaseUtil`
3. 新建 Hbase 配置对象 `new HBaseConfiguration()`，封装装置参数
4. 使用 `ConnectionFactory.createConnection(configuration)` 获取 hbase 连接
5. 使用 `connection.getAdmin` 获取与客户端的连接

6. 编写增删改查方法

- (2) 获取 Table 操作对象，方法名：getTable
- (3) 获取单列数据，方法名：getData
- (4) 存储单列数据，方法名：putData
- (5) 存储多列数据，方法名：putMapData
- (6) 删除数据，方法名：deleteData

API

方法名	用途	参数说明	返回值
getTable	创建/获取表	tableNameStr:表名 columnFamily:列族名	HBase Table 对象
getData	根据 rowkey，列族+列名获取数据	tableNameStr: 表名 rowkey:String rowkey columnFamily:列族名 column:列名	列对应的数据 String 类型
putData	插入/更新一列数据	tableNameStr: 表名 rowkey:String rowkey columnFamily:列族名 column:String 列名 data:String 列值	空
putMapData	插入多个列数据	tableNameStr: 表名 rowkey:String rowkey columnFamily:列族名 mapData:列/列值	空
deleteData	根据 rowkey 删除一条数据	tableNameStr: 表名 rowkey:rowkey columnFamily: 列族名	空

完整代码

```
package cn.itcast.utils

import cn.itcast.config.GlobalConfig
import org.apache.hadoop.hbase.client._
import org.apache.hadoop.hbase.util.Bytes
import org.apache.hadoop.hbase.{HBaseConfiguration, HColumnDescriptor, HTableDescriptor, TableName}

/**
```



```

* @Date 2019/5/29

*/
object HbaseUtil {

    private val configuration = new HBaseConfiguration()

    configuration.set("hbase.master", GlobalConfig.hmaster)
    configuration.set("hbase.zookeeper.quorum", GlobalConfig.hzk)
    configuration.set("hbase.zookeeper.property.clientPort", GlobalConfig.hPoint)
    configuration.set("hbase.rpc.timeout", GlobalConfig.hRpc)
    configuration.set("hbase.client.operation.timeout", GlobalConfig.hClientTimeout)
    configuration.set("hbase.client.scanner.timeout.period", GlobalConfig.hScanTimeout)

    private val connection: Connection = ConnectionFactory.createConnection(configuration)
    private val admin: Admin = connection.getAdmin

    //初始化 table 操作
    def getTable(tableName: String, colFm: String): Table = {
        val tbName: TableName = TableName.valueOf(tableName)
        //构建表描述器
        val nameDescriptor = new HTableDescriptor(tbName)
        //构建列族描述器
        val columnDescriptor = new HColumnDescriptor(colFm)
        nameDescriptor.addFamily(columnDescriptor)
        if (!admin.tableExists(tbName)) {
            admin.createTable(nameDescriptor)
        }
        val table: Table = connection.getTable(tbName)
        table
    }

    //查询单列数据
    def getData(tableName: String, colFm: String, col: String, rowKey: String): String = {
        val table = getTable(tableName, colFm)
        val get = new Get(Bytes.toBytes(rowKey))
        val result = table.get(get)
        val bytes = result.getValue(Bytes.toBytes(colFm), Bytes.toBytes(col))
        var str: String = null
        if (bytes != null && bytes.length > 0) {
            str = new String(bytes)
        }
        str
    }
}

```

```
//put 单列数据
```

```
def putData(tableName: String, colFm: String, col: String, data: String, rowKey: String): Unit = {  
    val table = getTable(tableName, colFm)  
    try {  
        val put = new Put(Bytes.toBytes(rowKey))  
        put.addColumn(Bytes.toBytes(colFm), Bytes.toBytes(col), Bytes.toBytes(data))  
        table.put(put)  
    } catch {  
        case e: Exception => e.printStackTrace()  
    } finally {  
        table.close()  
    }  
}
```

```
//put Map 数据
```

```
def putMapData(tableName: String, colFm: String, map: Map[String, Any], rowKey: String): Unit = {  
    val table = getTable(tableName, colFm)  
    try {  
        val put = new Put(Bytes.toBytes(rowKey))  
        for ((x, y) <- map) {  
            put.addColumn(Bytes.toBytes(colFm), Bytes.toBytes(x), Bytes.toBytes(y.toString))  
        }  
        table.put(put)  
    } catch {  
        case e: Exception => e.printStackTrace()  
    } finally {  
        table.close()  
    }  
}
```

```
//删除数据
```

```
def deleteData(tableName: String, colFm: String, rowKey: String): Unit = {  
    val table = getTable(tableName, colFm)  
    try {  
        val delete = new Delete(Bytes.toBytes(rowKey))  
        table.delete(delete)  
    } catch {  
        case e: Exception => e.printStackTrace()  
    } finally {  
        table.close()  
    }  
}
```

```
}
```

# TimeUtil 开发

获取线程安全的日期格式化对象:

FastDateFormat.getInstance(format)

```
package cn.itcast.utils

import java.util.Date
import org.apache.commons.lang3.time.FastDateFormat

/**
 * @Date 2019/5/29
 */
object TimeUtil {

    def getTime(timestamp:Long,format:String):String={
        val date: Date = new Date(timestamp)
        val format: FastDateFormat = FastDateFormat.getInstance(format)
        val str: String = format.format(date)
        str
    }
}
```

注意: FastDateFormat 是线程安全的

# 实时频道热点分析业务开发

## 业务介绍

频道热点，就是要统计频道被访问（点击）的数量。

分析得到以下的数据：

频道 ID	访问数量
频道 1	150
频道 2	231
频道 3	452

需要将历史的点击数据进行累加

## 步骤

1. 创建实时热点样例类，专门用来计算实时热点的数据

- 2. 将预处理后的数据， 转换 为要分析出来的数据（频道、访问次数）样例类
- 3. 按照 频道 进行分组（分流）
- 4. 划分时间窗口（3 秒一个窗口）
- 5. 进行合并计数统计
- 6. 将计算后的数据下沉到 Hbase

实现

- 1. 创建一个 ChannelHotTask 单例对象
- 2. 添加一个 ChannelHotTask 样例类，它封装要统计的两个业务字段：频道 ID（channelID）、访问数量（visited）
- 3. 在 ChannelHotTask 中编写一个 process 方法，接收预处理后的 DataStream
- 4. 使用 map 算子，将 Message 对象转换为 ChannelHotTask
- 5. 按照频道 ID 进行分流
- 6. 划分时间窗口（3 秒一个窗口）
- 7. 执行 reduce 合并计算
- 8. 将合并后的数据写入 hbase

判断 hbase 中是否已经存在结果记录  
若存在，则获取后进行累加  
若不存在，则直接写入

实时频道 PV/UV 分析

针对频道的 PV、UV 进行不同时间维度的分析。有以下三个维度：

- 小时
- 天
- 月

我们会分别来实现不同维度的分析。

业务介绍

PV(访问量)

即 Page View，页面刷新一次算一次。

UV(独立访客)

即 Unique Visitor，指定时间内相同的客户端只被计算一次

统计分析后得到的数据如下所示：

频道 ID	时间	PV	UV
频道 1	2017010116	1230	350
频道 2	2017010117	1251	330
频道 3	2017010118	5512	610

# 实时频道用户新鲜度分析

## 业务介绍

用户新鲜度即分析网站每小时、每天、每月活跃的新老用户占比

可以通过新鲜度：

从宏观层面上了解每天的新老用户比例以及来源结构

当天新增用户与当天 **推广行为** 是否相关

...

统计分析要得到的数据如下：

频道 ID	时间	新用户	老用户
频道 1	201703	512	144
频道 1	20170318	411	4123
频道 1	2017031810	342	4412

# 实时频道地域分析业务开发

## 业务介绍

通过地域分析，可以帮助查看地域相关的 PV/UV、用户新鲜度。

需要分析出来指标

PV

UV

新用户

老用户

需要分析的维度

地域（国家省市）——这里为了节省时间，只分析市级的地域维度，其他维度大家可以自己来实现

时间维度（时、天、月）

统计分析后的结果如下：

频道 ID	地域（国/省/市）	时间	PV	UV	新用户	老用户
频道 1	中国北京市 朝阳区	201809	1000	300	123	171

频道 1	中国北京市 朝阳区	20180910	512	123	23	100
频道 1	中国北京市 朝阳区	2018091010	100	41	11	30

## 实时运营商分析业务开发

### 业务介绍

分析出来中国移动、中国联通、中国电信等运营商的指标。来分析，流量的主要来源是哪个运营商的，这样就可以进行较准确的网络推广。

需要分析出来指标：

新用户

老用户

需要分析的维度

运营商

时间维度（时、天、月）

统计分析后的结果如下：

频道 ID	运营商	时间	新用户	老用户
频道 1	中国移动	201809	0	300
频道 1	中国联通	20180910	0	1
频道 1	中国电信	2018091010	2	0

## 实时频道浏览器分析业务开发

### 业务介绍

需要分别统计不同浏览器（或者客户端）的占比

需要分析出来指标

PV

UV

新用户

老用户

需要分析的维度

浏览器

时间维度（时、天、月）

统计分析后的结果如下：

频道 ID	浏览器	时间	PV	UV	新用户	老用户
频道 1	360 浏览器	201809	1000	300	0	300
频道 1	IE	20180910	123	1	0	1
频道 1	Chrome	2018091010	55	2	2	0

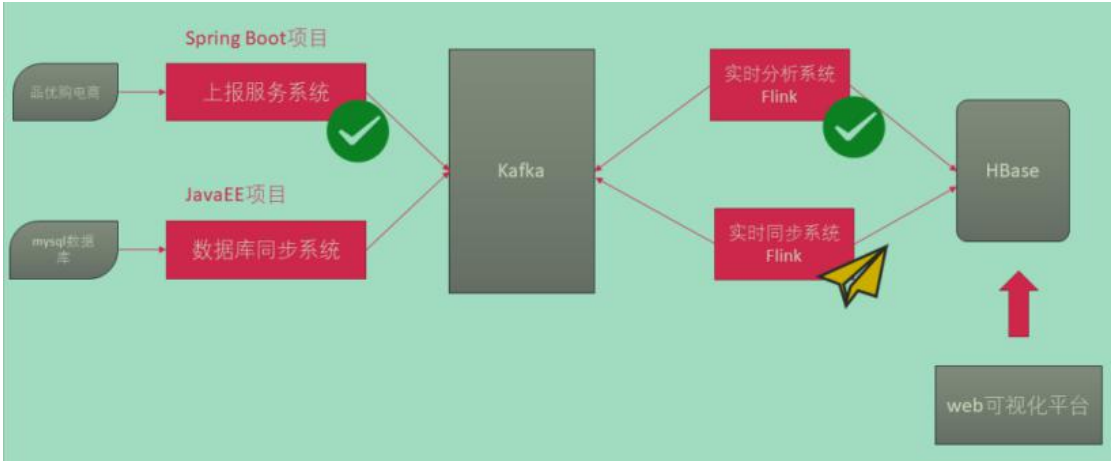
## 开发业务数据库同步系统

## 实时数据同步系统目标

理解 canal 数据同步解决方案

安装 canal

实现 Flink 数据同步系统



需求分析

业务背景

企业运维的数据库最常见的是 mysql；但是 mysql 有个缺陷：当数据量达到千万条的时候，mysql 的相关操作会变的非常迟缓；

如果这个时候有需求需要实时展示数据；对于 mysql 来说是一种灾难，甚至会导致 mysql 宕机。

要进行海量数据分析，就需要将 mysql 中的数据 **同步** 到其他的海量数据存储介质（HDFS、hbase）中。那如何来导出呢？

#### 方案一

mysql → logstash → kafka → sparkStreaming → hbase → web

#### 方案二

mysql → sqoop → hbase → web

但是无论使用 logstash 还是使用 sqoop，都避免不了一个尴尬的问题：

他们在导出数据过程中需要去 mysql 中做查询操作：

比如 logstash：

```
input {
  stdin { }
  jdbc {
    jdbc_driver_library => "G:/MvnRepository/mysql/mysql-connector-java/5.1.41/mysql-connector-java-5.1.41.jar"
    jdbc_driver_class => "com.mysql.jdbc.Driver"
    jdbc_connection_string => "jdbc:mysql://localhost:3306/xnc"
    jdbc_user => "root"
    jdbc_password => "123456"
    # or jdbc_password_filepath => "/path/to/my/password_file"
    # where p.update_time >= :sql_last_start
    statement => "SELECT id, product_spec_id,zone_id,recorded_by,CAST(price_per_unit AS CHAR) price_per_unit,uom,latest,recording_date,create_time,update_time from price p"
    jdbc_paging_enabled => "true"
    jdbc_page_size => "50000"
  }
}
```

比如 sqoop：

```
[hdfs@node196 bin]$ sqoop eval --connect jdbc:mysql://192.168.180.11/angel --username anqi -password anqi_mima \
--query "SELECT xi.*,jing.name,wang.latitude,wang.longitude \
FROM xi ,jing, wang \
WHERE xi.id=jing.foreignId AND wang.id=xi.id AND xi.date>='2015-09-01' AND xi.date<='2015-10-01'"
```

方案一和方案二都不可避免的，都需要去 sql 中查询出相关数据，然后才能进行同步；这样对于 mysql 来说本身就是增加负荷操作；

#### 方案三

mysql → canal(binlog) → kafka → flink → hbase

Canal 可以实时解析 mysql 的 binlog 日志，通过读取 binlog 日志，将数据输出到 Kafka。不需要执行 SQL 语句，不会增加 mysql 压力

### Canal 数据采集

#### 创建 mysql 业务表

mysql 脚本： day04\资料\mysql 脚本 目



```

=====商品表=====

create table commodity(
commodityId int(10) , #商品 ID
commodityName varchar(20), #商品名称
commodityTypeId int(20), #商品类别 ID
originalPrice double(16,2), #原价
activityPrice double(16,2) #活动价 );

=====类别表=====

create table category(
categoryId int(10), #商品类别 ID
categoryName varchar(20), #商品类别名称
categoryGrade int(20) #商品类别等级 );

=====商家店铺表=====

create table merchantStore(
merchantId int(20) , #商家 ID
storeName varchar(20), #商家店铺名称
storeId varchar(20) #商家店铺 ID );

```

## 开启 mysql 的 binlog 日志

Mysql 的 binlog 日志作用是用来记录 mysql 内部增删等对 mysql 数据库有更新的内容的记录（对数据库的改动），对数据库的查询 select 或 show 等不会被 binlog 日志记录;主要用于数据库的主从复制以及增量恢复。

### mysql 的 binlog 日志必须打开 log-bin 功能才能生成 binlog

```

-rw-rw---- 1 mysql mysql 669 11月 10 21:29 mysql-bin.000001
-rw-rw---- 1 mysql mysql 126 11月 10 22:06 mysql-bin.000002
-rw-rw---- 1 mysql mysql 11799 11月 15 18:17 mysql-bin.000003

```

### 修改/etc/my.cnf, 在里面添加如下内容

```

[mysqld]
log-bin=/var/lib/mysql/mysql-bin 【默认 binlog 日志存放路径, mysql-bin 日志名称】
binlog-format=ROW 【日志中会记录成每一行数据被修改的形式】
server_id=1 【指定当前机器的服务 ID（如果是集群，不能重复）】

```

配置完毕之后，登录 mysql，输入如下命令：

```
show variables like 'log_bin'
```

出现如下形式，代表 binlog 开启：

```
mysql> show variables like '%log_bin%' ;
```

Variable_name	Value
log_bin	ON
log_bin_trust_function_creators	OFF
log_bin_trust_routine_creators	OFF
sql_log_bin	ON

4 rows in set (0.00 sec)

#### 四、获取 binlog 文件列表

```
mysql> show binary logs;
```

```
mysql> show binary logs;
+-----+-----+
| Log_name | File_size |
+-----+-----+
| mysql-bin.000001 | 120 |
+-----+-----+
1 row in set (0.00 sec)
```

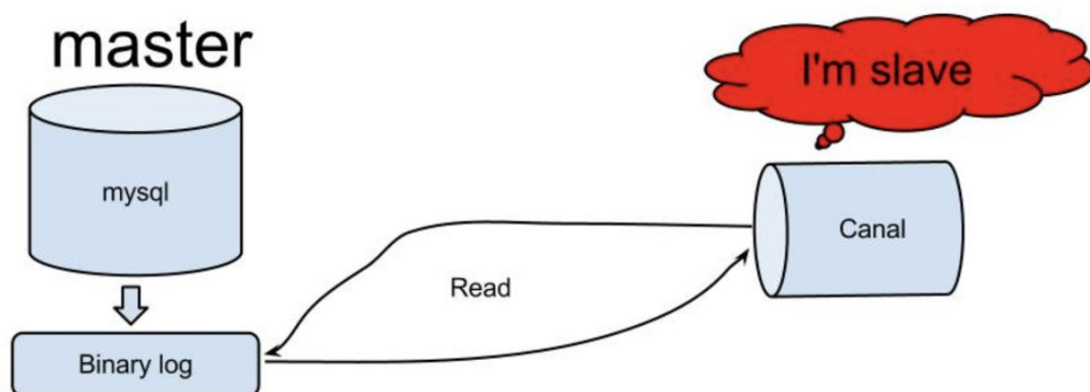
#### 安装 canal

#### Canal 介绍

canal 是阿里巴巴旗下的一款开源项目，纯 Java 开发。基于数据库增量日志解析，提供增量数据订阅&消费，目前主要支持了 MySQL（也支持 mariadb）。

起源：早期，阿里巴巴 B2B 公司因为存在杭州和美国双机房部署，存在跨机房同步的业务需求。不过早期的数据库同步业务，主要是基于 trigger 的方式获取增量变更，不过从 2010 年开始，阿里系公司开始逐步的尝试基于数据库的日志解析，获取增量变更进行同步，由此衍生出了增量订阅&消费的业务，从此开启了一段新纪元。

#### 工作原理



原理相对比较简单：

- 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)

## 解压安装包

```
tar -zxvf canal.deployer-1.0.23.tar.gz -C /export/servers/canal
```

修改配置文件:

```
vim /export/servers/canal/conf/example/instance.properties
```

```
## mysql serverId
canal.instance.mysql.slaveId = 1234

# position info
canal.instance.master.address = 192.168.52.101:3306
canal.instance.master.journal.name =
canal.instance.master.position =
canal.instance.master.timestamp =

#canal.instance.standby.address =
#canal.instance.standby.journal.name =
#canal.instance.standby.position =
#canal.instance.standby.timestamp =

# username/password
canal.instance.dbusername = root
canal.instance.dbpassword = 123456
canal.instance.defaultDatabaseName =
canal.instance.connectionCharset = UTF-8

# table regex
canal.instance.filter.regex = .*\\..*
# table black regex
canal.instance.filter.black.regex =
```

## 编写 canal 代码

仅仅安装了 canal 是不够的; canal 从架构的意义上来说相当于 mysql 的“从库”，此时还并不能将 binlog 解析出来实时转发到 kafka 上，因此需要进一步开发 canal 代码;

Canal 已经帮我们提供了示例代码，只需要根据需求稍微更改即可;

Canal 提供的代码:

```
https://github.com/alibaba/canal/wiki/ClientExample
```

上面的代码中可以解析出 binlog 日志，但是没有将数据落地到 kafka 的代码逻辑，所以我们还需要添加将数据落地 kafka 的代码;

Maven 导入依赖:

```
<dependency>
  <groupId>com.alibaba.otter</groupId>
  <artifactId>canal.client</artifactId>
  <version>1.0.23</version>
</dependency>

<!-- https://mvnrepository.com/artifact/org.apache.kafka/kafka -->
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka_2.11</artifactId>
```

```
<version>0.9.0.1</version>
</dependency>
```

## 测试 canal 代码

### 1、 启动 kafka 并创建 topic

```
/export/servers/kafka/bin/kafka-server-start.sh /export/servers/kafka/config/server.properties >/dev/null
2>&1 &

/export/servers/kafka/bin/kafka-topics.sh --create --zookeeper hadoop01:2181 --replication-factor 1
--partitions 1 --topic mycanal
```

### 2、 启动 mysql 的消费者客户端，观察 canal 是否解析 binlog

```
/export/servers/kafka/bin/kafka-console-consumer.sh --zookeeper hadoop01:2181 --from-beginning --topic mycanal
```

### 2、启动 mysql: service mysqld start

### 3、启动 canal: canal/bin/startup.sh

### 4、进入 mysql: mysql -u 用户 -p 密码; 然后进行增删改

## 编写 kafka 代码

### 配置:

```
prop.put("zookeeper.connect", "node01:2181,node02:2181,node03:2181");
prop.put("metadata.broker.list", "node01:9092,node02:9092,node03:9092");
prop.put("serializer.class", StringEncoder.class.getName());
```

### 代码:

```
public class KafkaProducer {

    public static Properties getKafka() {

        Properties prop = new Properties();

        prop.put("zookeeper.connect", "node01:2181,node02:2181,node03:2181");
        prop.put("metadata.broker.list", "node01:9092,node02:9092,node03:9092");
        prop.put("serializer.class", StringEncoder.class.getName());

        return prop;
    }

    public static Producer getProducer() {

        ProducerConfig producerConfig = new ProducerConfig(getKafka());
        Producer<Integer, String> producer = new Producer<Integer, String>(producerConfig);

        return producer;
    }

    public static void senMsg(String topic, String key, String data) {
```

```
        Producer producer = getProducer();

        producer.send(new KeyedMessage(topic, key, data));

    }

}
```

## Flink 实时数据同步系统开发

### binlog 日志格式分析

#### 测试日志数据

```
{
  "emptyCount": 2,
  "logFileName": "mysql-bin.000002",
  "dbName": "pyg",
  "logFileOffset": 250,
  "eventType": "INSERT",
  "columnValueList": [{
    "columnName": "commodityId",
    "columnValue": "1",
    "isValid": "true"
  },
  {
    "columnName": "commodityName",
    "columnValue": "耐克",
    "isValid": "true"
  },
  {
    "columnName": "commodityTypeId",
    "columnValue": "1",
    "isValid": "true"
  },
  {
    "columnName": "originalPrice",
    "columnValue": "888.0",
    "isValid": "true"
  },
  {
    "columnName": "activityPrice",
    "columnValue": "820.0",
    "isValid": "true"
  }
}
```

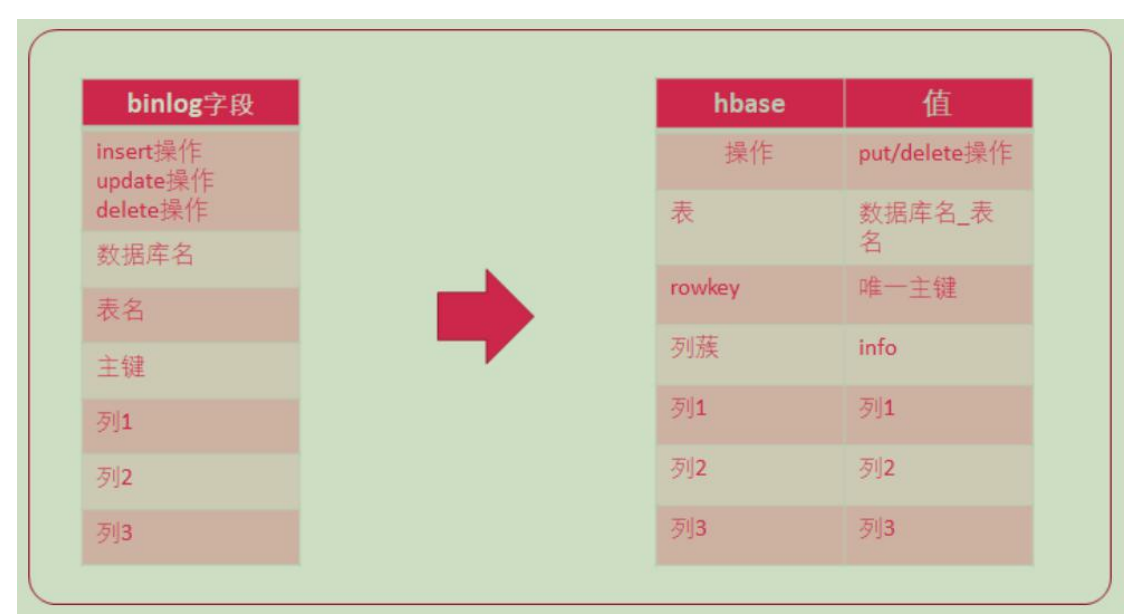
```
],
  "tableName": "commodity",
  "timestamp": 1553741346000
}
```

格式分析

字段以及说明

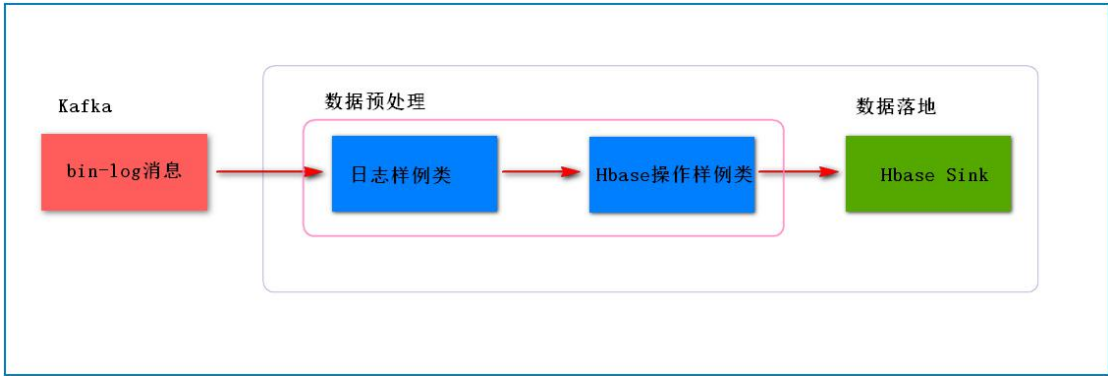
字段名称	说明	示例
emptyCount	操作序号（第几条记录）	12
logFileName	binlog 文件名	mysql-bin.000001
dbName	数据库名称	pyg
logFileOffset	binlog 文件偏移位置	100
eventType	操作类型	INSERT 或 UPDATE 或 DELETE
columnValueList	列值列表	{ "columnName": "列名", "columnValue": "列值", "isValid": "是否有效" }
tableName	表名	commodity
timestamp	执行时间戳	1553701139000

数据同步说明



要确保 hbase 中的 rowkey 是唯一的，数据落地 不能被覆盖

数据处理



1. Flink 对接 Kafk
2. 对数据进行预处理
3. 将数据落地到 hbase

Flink 程序开发

创建目录

在 sync-db 项目的 scala 目录中中，创建以下包结构：

包名	说明
com.itcast.task	存放所有 flink 任务代码
com.itcast.util	工具类
com.itheima.bean	存放实体类

初始内容搭建

1. 将 day04\资料\数据同步\配置 目录的 pom.xml 文件中的依赖导入到 sync-db 项目的 pom.xml
2. sync-db 模块添加 scala 支持
3. main 创建 scala 文件夹，并标记为源代码目录
4. 将 day04\资料\数据同步\配置 目录中的 application.properties 和 log4j.properties 配置文件，导入到 resource 目录
5. 复制之前 Flink 项目中的 GlobalConfigUtil 和 HBaseUtil

定义原始 Canal 消息 样例类

步骤

1. 在 bean 包下创建 Canal 原始消息 映射样例类
2. 在 Cannal 样例类 中编写 apply 方法，使用 FastJSON 来解析数据，将字段获取，创建 Cannal 样例类对象
3. 编写 main 方法测试是否能够成功构建样例类对象

参考代码

```
case class Canal(var emptyCount: Long,
```

```

        var logFileName: String,

        var dbName: String,

        var logFileOffset: Long,

        var eventType: String,

        var columnValueList: String,

        var tableName: String,

        var timestamp: Long)

object Canal {

    def apply(json: String): Canal = {

        val jsonObject = JSON.parseObject(json)

        Canal(

            jsonObject.getString("emptyCount").toString.toLong,

            jsonObject.getString("logFileName").toString,

            jsonObject.getString("dbName").toString,

            jsonObject.getString("logFileOffset").toString.toLong,

            jsonObject.getString("eventType").toString,

            jsonObject.getString("columnValueList").toString,

            jsonObject.getString("tableName").toString,

            jsonObject.getString("timestamp").toString.toLong

        )

    }

}

```

## 解析 Kafka 数据流为 Canal 样例类

### 步骤

1. 在 `map` 算子将消息转换为 `Canal 样例类` 对象
2. 打印测试，如果能输出以下信息，表示成功

```

Canal(mysql-bin.000002,1893,pyg,commodity,DELETE,
[{"isValid":"false","columnValue":"5","columnName":"commodityId"}, {"isValid":"false","columnValue":"索菲亚","columnName":"commodityName"}, {"isValid":"false","columnValue":"3","columnName":"commodityTypeId"}, {"isValid":"false","columnValue":"35000.0","columnName":"originalPrice"}, {"isValid":"false","columnValue":"30100.0","columnName":"activityPrice"}],3,1553743567000)
HBaseOperation(DELETE,commodity,info,5,,)

```

## 添加水印支持

### 步骤

1. 使用 Canal 中的 `timestamp` 字段，生成水印数据
2. 重新运行 Flink，打印添加水印后的数据

## 定义 HbaseOperation 样例类

HbaseOperation 样例类主要封装对 Hbase 的操作，主要封装以下字段：

操作类型（opType） = INSERT/DELETE/UPDATE



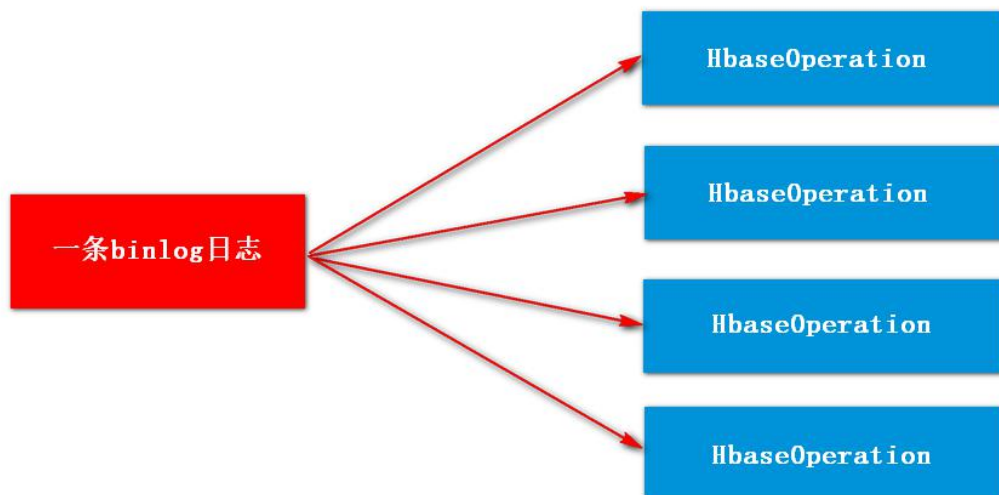
表名 (tableName) = 数据库名\_表名  
列族名 (cfName) = 固定为 info  
rowkey = 唯一主键 (去 binlog 中列数据的第一个)  
列名 (colName) = binlog 中列名  
列值 (colValue) = binlog 中列值

#### 参考代码

```
/**
 * HBase 操作样例类
 *
 * @param opType 操作类型
 * @param tableName 表名
 * @param cfName 列族名
 * @param rowkey rowkey 唯一主键
 * @param colName 列名
 * @param colValue 列值
 */
case class HbaseOperation(var opType: String,
    var tableName: String,
    var cfName: String,
    var rowkey: String,
    var colName: String,
    var colValue: String)
```

#### 将 Canal 样例类转换为 HbaseOperation 样例类

一个 binlog 消息中，会有多个列的操作。它们的映射关系如下：



我们可以使用 flatMap 算子，来生成一组 HbaseOperation 操作

#### 步骤

1. 创建一个预处理任务对象
2. 使用 flatMap 对水印数据流转换为 HbaseOperation  
根据 eventType 分别处理 HbaseOperation 列表

生成的表名为 数据库名\_表名

rowkey 就是第一个列的值

INSERT 操作 -> 将所有列值转换为 HbaseOperation

UPDATE 操作 -> 过滤掉 isValid 字段为 false 的列，再转换为 HBaseOperation

DELETE 操作 -> 只生成一条 DELETE 的 HbaseOperation 的 List

序号	值	说明
eventType	INSERT	表示这是一个插入操作
columnValueList	JSON 数组	columnValueList 中，isValid 总是为 true

#### INSERT 操作记录

#### UPDATE 操作记录

序号	值	说明
eventType	UPDATE	表示这是一个更新操作
columnValueList	JSON 数组	columnValueList 中，更新的 isValid 字段为 true，未更新的为 false

#### DELETE 操作记录

序号	值	说明
eventType	DELETE	表示这是一个删除操作
columnValueList	JSON 数组	columnValueList 中，isValid 总是为 false

#### 实现

1. 在 task 包下创建 PreprocessTask 单例对象，添加 process 方法
2. 使用 flatMap 对 Canal 样例类进行扩展
3. 使用 JSON 解析 Canal 样例类中的 列值列表数据，并存储到一个 List 中
4. 遍历 List 结构，构建 HbaseOperation 样例类对象
5. 打印测试
6. 启动 Flink 验证程序是否正确处理

#### 参考代码

```
package cn.itcast

import cn.itcast.bean.{Canal, HbaseOperation}
import com.alibaba.fastjson.{JSON, JSONArray, JSONObject}
import org.apache.flink.streaming.api.scala.{DataStream, _}

import scala.collection.mutable

/**
 * @Date 2019/5/30
 */
object PreprocessTask {
```

```

case class ColumnValuePair(columnName: String, columnValue: String, isValid: Boolean)

def process(waterData: DataStream[Canal]): DataStream[HabseOperation] = {
  val result: DataStream[HabseOperation] = waterData.flatMap {
    line =>
      val opsType = line.eventType
      val tabName = s"mysql.${line.dbName}.${line.tableName}"
      //组织hbase
      val colFm = "info"
      val columnValueList: List[ColumnValuePair] = parseToColumnPair(line.columnValueList)
      val rowkey = columnValueList(0).columnValue
      //将所有列转换为HabseOperation
      opsType match {
        case "INSERT" => {
          columnValueList.map {
            line =>
              HabseOperation(opsType, tabName, colFm, rowkey, line.columnValue, line.columnName)
          }
        }
        case "UPDATE" => {
          columnValueList.filter(_.isValid).map(
            line => {
              HabseOperation(opsType, tabName, colFm, rowkey, line.columnValue, line.columnName)
            }
          )
        }
        case "DELETE" => {
          List(HabseOperation(opsType, tabName, colFm, rowkey, "", ""))
        }
      }
  }
  //打印测试
  result.print()
  result
}

def parseToColumnPair(columnValueList: String): List[ColumnValuePair] = {
  val array: JSONArray = JSON.parseArray(columnValueList)
  val columnValuePairs = new mutable.ListBuffer[ColumnValuePair]
  for (i <- 0 until array.size()) {
    val jsonObject: JSONObject = array.getJSONObject(i)
    columnValuePairs += ColumnValuePair(jsonObject.getString("columnName"),
      jsonObject.getString("columnValue"),
      jsonObject.getBoolean("isValid"))
  }
}

```

```

    )
  }

  columnValuePairs.toList
}

}

```

## Flink 数据同步到 hbase

### 步骤

1. 分两个落地实现，一个是 **delete**，一个是 **insert/update**（因为 hbase 中只有一个 put 操作，所以只要是 insert/update 都转换为 put 操作）

2. 启动 **hbase**

3. 启动 **flink** 测试

### 参考代码

```

val operationDataStream: DataStream[HBaseOperation] = PreprocessTask.process(watermarkData)

operationDataStream.addSink(new SinkFunction[HBaseOperation] {

  override def invoke(value: HBaseOperation): Unit = {

    value.opType match {

      case "DELETE" =>

        HBaseUtil.deleteData(value.tableName, value.rowkey, value.cfName)

      case _ => // INSERT/UPDATE

        HBaseUtil.putData(value.tableName, value.rowkey, value.cfName, value.colName, value.colValue)

    }

  }

})

```

## 验证 Flink 同步数据功能

### 步骤

1. 启动 **mysql**
2. 启动 **canal**
3. 启动 **zookeeper** 集群
4. 启动 **kafka** 集群
5. 启动 **hdfs** 集群
6. 启动 **hbase** 集群
7. 启动 **Flink 数据同步程序**
8. 启动 **Canal 数据同步程序**
9. 在 mysql 中执行 insert、update、delete 语句，查看 **hbase** 数据是否落地





