



**本科毕业设计**

题 目 短视频分享应用及后台管理系统

专 业 软件工程

作者姓名 孟贺

学 号 2016205645

单 位 计算机学院

指导教师 赵传申

**2020 年 5 月**

**教务处编**

原创性声明

本人郑重声明：所提交的学位论文是本人在导师指导下，独立进行研究取得的成果。除文中已经引用的内容外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得聊城大学或其他教育机构的学位证书而使用过的材料。对本文的研究作出重要贡献的个人和集体，均在文中以明确的方式表明。本人承担本声明的相应责任。

学位论文作者签名： 日期：

指 导 教 师 签 名: 日期：

**摘 要**

本实时大数据计算平台设计的背景是随着业务量的迅速增加，数据量也随之提升，此时使用RDBMS（关系型数据库）已经收效甚微，后端业务关系数据库已经做到分库分表，随着分库分表缓解后端业务数据库的压力的同时，开发变得越来越复杂，要想获取整个平台的客户充值情况更是需要做出对后端程序的一定的代码入侵，并且计算时长成倍的增长。非集群机器计算由于现代的大数据量过大无法存储更不用说对它进行计算了，非集群机器处理成为大数据计算和大数据存储的瓶颈，因此我们这里基于大数据组件进行开发，使用计算机集群对数据进行存储和计算，将最终的结果存储到RDBMS中，方便对接现代的WEB后端程序展示或导出供数据分析师进行分析。

实时大数据计算平台设计的意义是满足数据分析师的快速掌控充值系统概况的需求，方便数据分析师和业务产品经理随时了解充值系统的收入指出以及订单的创建取消等。本课题的设计工作分三步进行，首先是技术选型，随后是该技术的文档以及文献参阅，接着是功能设计，最后是结果数据准确性的测试与校验。

本设计基于Spark Streaming作为实时计算引擎开发，使用Kafka作为中间件，使用MYSQL作为计算平台元数据数据库，HBase作为结果数据库，Redis作为中间结果数据库，设计并实现一款高性能的实时计算平台，本设计主要提供接口和结果数据，结果数据面向数据分析师和业务产品经理两个职位。

**关键词：**Hadoop；Spark Streaming；Kafka 5个

**Abstract**

The background of this real-time big data computing platform design is that as the volume of business grows rapidly, the amount of data will also increase. At this point, the use of RDBMS (relational database) has little effect, and the back-end business relational database has been divided into sub-libraries. Table, as the sub-library table eases the pressure on the back-end business database, data streams becomes more and more complicated. In order to obtain the customer recharge of the entire platform, it is necessary to make certain code invade the back-end program. And the calculation time is multiplied. Original independent calculations are not suitable for handling large amounts of modern data. Single-machine processing becomes a bottleneck for computing resources and storage resources. Therefore, we are here based on big data component development becomes more and more complexed, using computer clusters to store and calculate data, and the end result will be the end result. It is stored in the RDBMS to facilitate the docking of modern WEB backend programs for display or export for analysis by data analysts.

The significance of the real-time big data computing platform design is to meet the data analyst's need to quickly control the recharge system configuration file, so that data analysts and business product managers can keep abreast of the recharge system's revenue indication and cancellation commands. The design work for this topic is done in three steps. The first is technology choice, then technical documentation and literature reference, followed by the functional design of the real-time computing platform, and finally the data result test verification of the real-time computing platform.

The Spark Streaming was used as a real-time calculation engine for this project. It uses Kafka as a middleware, MYSQL as a computing platform metabase, HBase(Hadoop database) as a result database, and Redis as an intermediate result database to design and implement a high-performance real-time computing platform. The design primarily provides interface and result data, and the resulting data applies to data analysts and business product managers.

**Key words:** Hadoop; Spark Streaming; Kafka

# 短视频分享应用及后台管理系统

# 1.引言

## 1.1 研究目的和研究意义

类比传统的局限于文字图片类的社交软件，短视频更加生动有趣，而随着4G/5G的普及也使得观看短视频更加流畅方便，用户观看体验更好。同时在生活节奏不断加快的都市生活中，短视频不需要大量占用整段的时间，更适合碎片化的时间分布，也给紧凑的工作生活带来简短而有效的休闲放松。基于这样的原因，我希望开发并分析一个短视频应用的整个建设过程，无论是对自己的提升还是帮助别人学习，都是很有益处的。

## 1.2 研究现状和发展趋势

短视频平台化发展趋势明显，各大龙头平台瓜分市场，激烈竞争。最早出现的秒拍、小咖秀及美拍等短视频APP作为奠定用户基础的第一代应用，为短视频后期的井喷式爆发奠定了前期用户基础。秒拍与微博合作，重视明星、网红、KOL的引领作用，快手、抖音迅速崛起，成为当前短视频领域的两大寡头，各大互联网巨头也围绕短视频领域展开争夺，梨视频、我们视频、七环视频纷纷崛起。以快手为例，这个以短视频为核心传播内容的社交平台内聚集了大量三四线城市以及农村用户，为平台带来巨额流量。

## 1.3 本文研究内容

本系统是一个基于springboot的短视频分享应用及后台管理系统，后端采springboot,hls,ffmpe等主流技术幵发，移动端采用Flutter实现，Flutter是谷歌的移动UI框架，可以快速在iOS和Android上构建高质量的原生用户界面。现有的开发技术已非常成熟，且被广泛应用于各行各业，利用现有技术完全可以达到功能目标。相关技术社区十分活跃，遇到问题也能及时解决。考虑开发期限较为充裕，预计可以在规定的时间内完成开发。

# 2. 开发语言与技术简介

## 2.1 Java简介

Java是一种基于类、面向对象的编程语言。Java的目的是让应用程序开发人员编写一次，在任何地方运行（WORA），这意味着编译过的Java代码可以在所有支持Java的平台上运行，而无需重新编译。Java应用程序通常被编译成字节码，可以在任何Java虚拟机（JVM）上运行，而不管底层的计算机架构如何。java的语法类似于C++和C++，但它们的底层设备比它们中的任何一个都要低。截至2019年，根据GitHub的说法，Java是最流行的编程语言之一，特别是用于客户机-服务器web应用程序，据报道有900万开发人员。

## 2.2 Spring 简介

IntelliJ IDEA是一个收费的一系列的开发工具之首的开发工具，不仅适用于JVM且功能强大并符合人体工程学的IDE（Integrated Development Environment），高级版本支持多种语言，更有多种强大的功能使得写代码变得更容易，例如：自动选取、导航模式、JUnit的完美支持、Debug、融合Git和SVN等的代码版本控制功能。

## 2.3 Dart和Flutter 简介

Dart是一种针对多平台应用程序客户端开发优化编程语言。它由Google开发，用于构建移动、桌面、服务器和web应用程序。Dart是面向对象、基于类、垃圾收集的语言，具有C语言风格的语法。Dart可以编译为机器语言或JavaScript。它支持接口、混合、抽象类、具体化泛型和类型推断。

Flutter是一个基于Dart、由[谷歌](https://zh.wikipedia.org/wiki/%E8%B0%B7%E6%AD%8C)开发的[开源](https://zh.wikipedia.org/wiki/%E5%BC%80%E6%BA%90%E8%BD%AF%E4%BB%B6)移动应用软件开发工具包，用于为[Android](https://zh.wikipedia.org/wiki/Android)、[iOS](https://zh.wikipedia.org/wiki/IOS" \o "IOS)、 Windows、Mac、Linux、[Google Fuchsia](https://zh.wikipedia.org/wiki/Google_Fuchsia" \o "Google Fuchsia)开发应用[[4]](https://zh.wikipedia.org/wiki/Flutter" \l "cite_note-4)。Flutter是通过组织、创建不同的组件完成[用户界面设计](https://zh.wikipedia.org/wiki/%E7%94%A8%E6%88%B7%E7%95%8C%E9%9D%A2%E8%AE%BE%E8%AE%A1" \o "用户界面设计)的。在Flutter中，一个组件代表用户界面中[不可变](https://zh.wikipedia.org/wiki/%E4%B8%8D%E5%8F%AF%E8%AE%8A%E7%89%A9%E4%BB%B6" \o "不可变对象)的一部分；包括文本、多边形以及动画在内的所有图形都是用组件创建的。复杂的组件由简单的组件结合而成。Flutter的引擎主要使用[C++](https://zh.wikipedia.org/wiki/C%2B%2B" \o "C++)开发，通过Google的[Skia图形库](https://zh.wikipedia.org/wiki/Skia_Graphics_Library" \o "Skia Graphics Library)提供底层渲染支持，亦提供平台相关的[SDK](https://zh.wikipedia.org/wiki/%E8%BD%AF%E4%BB%B6%E5%BC%80%E5%8F%91%E5%B7%A5%E5%85%B7%E5%8C%85)，例如[Android](https://zh.wikipedia.org/wiki/Android" \o "Android)和[iOS](https://zh.wikipedia.org/wiki/IOS" \o "IOS)[[8]](https://zh.wikipedia.org/wiki/Flutter#cite_note-:0-8)。Flutter引擎是用于托管Flutter应用的可移植的运行环境。它实现了Flutter的核心库，包括动画和图形、文件和网络I/O、可访问性支持、插件架构以及Dart运行环境和编译工具链。

## 2.4 HLS和Ffmpeg 简介

**HTTP Live Streaming**（缩写是**HLS**）是由[苹果公司](https://zh.wikipedia.org/wiki/%E8%8B%B9%E6%9E%9C%E5%85%AC%E5%8F%B8" \o "苹果公司)提出基于[HTTP](https://zh.wikipedia.org/wiki/HTTP)的[流媒体](https://zh.wikipedia.org/wiki/%E6%B5%81%E5%AA%92%E4%BD%93" \o "流媒体)[网络传输协议](https://zh.wikipedia.org/wiki/%E7%BD%91%E7%BB%9C%E4%BC%A0%E8%BE%93%E5%8D%8F%E8%AE%AE)。它的工作原理是把整个流分成一个个小的基于HTTP的文件来下载，每次只下载一些。当媒体流正在播放时，客户端可以选择从许多不同的备用源中以不同的速率下载同样的资源，允许流媒体会话适应不同的数据速率。在开始一个流媒体会话时，客户端会下载一个包含元数据的[extended M3U (m3u8)](https://zh.wikipedia.org/w/index.php?title=Extended_M3U&action=edit&redlink=1" \o "Extended M3U（页面不存在）) [playlist](https://zh.wikipedia.org/w/index.php?title=Playlist&action=edit&redlink=1)文件，用于寻找可用的媒体流。FFmpeg是一个免费的开放源代码项目，包含用于处理视频，音频以及其他多媒体文件和流的库和软件。 FFmpeg专门用于基于命令行的视频和音频文件处理，并且广泛用于格式转码，基本编辑（修剪和串联），视频缩放，视频后期制作效果。

# 3.需求分析

## 3.1 平台可行性分析

### 3.1.1 技术可行性分析

（1）硬件可行性分析：硬件需求上需要最少三台Linux内核的机器（每台至少有1个core、1GB内存、50GB硬盘）作为集群的基础确保集群满足最低配置要求；使用虚拟机实现，三台机器需要互相设置 SSH免密码登陆、搭建基础的HDFS（Hadoop Distributed File System）、YARN（Yet Another Resource Negotiator）资源管理器、Zookeeper、Kafka、Spark、Scala、HBase以及单机的Redis。

（2）软件可行性分析： 通过使用开源大数据组件搭建可靠数据处理平台从而保证能够实现数据的完整性和一致性。

### 3.1.2 数据准确性分析

在Kafka数据消费Offset设计上使用Spark Task的分区ID以及批次数据到达时间作为唯一ID用以更新Kafka消费数据的元数据库数据。这样做的目的是实现Idempotent updates（幂等更新，多次写入不会对系统最终结果造成影响），使用唯一ID更新外部数据系统，查询到外部系统不存在该id的数据则提交原子性更新或插入到外部数据系统，否则跳过更新，以此保证程序失败后重启程序后数据出现重复读取的情况下不会对系统造成最终结果的影响。

|  |
| --- |
| dstream.foreachRDD { (rdd, time) =>  rdd.foreachPartition { partitionIterator =>  val partitionId = TaskContext.get.partitionId()  val uniqueId = generateUniqueId(time.milliseconds, partitionId)  // TODO use this uniqueId to transactionally commit the data in partitionIterator  } } |

### 3.1.3法律可行性分析

平台使用开源组件搭建，不涉及任何版权问题，在数据存储时间上也有严格按照法律规定存储删除数据。

平台可靠性较高，保证了数据的一致性、完整性，从而保证了计算出的数据的价值。此平台由于没有可视化操作界面可移植性较差，网络环境、磁盘环境等都需要专业人员操作，平台搭建完成后运行程序会将数据写入到MySQL、Redis、HBase，之后将几种数据库的读权限开放给其他开发人员即可，这样减少平台使用的复杂度，使得平台搭建和平台数据应用任务分离。

## 3.2 功能需求分析

### 3.2.1全网统计充值订单总金额和订单量

将从kafka中拉取的一个批次的数据和历史数据进行累加，这里因为是实时计算因此将历史数据存在Redis中。Redis中保存的数据有充值订单数量，订单充值全局平均请求方法调用时间，统计的充值成功订单金额与订单量等。

用于实时统计历史累计营业额，为上层决策作数据支持。

### 3.2.2每小时各个省份的充值失败数据量

统计个省份充值情况，存储在MySQL中，MySQL的数据有成功订单量，成功订单金额，总订单量等。

实时统计各个省份营业额，用于上层管理的战略调整，发展倾向。

### 3.2.3订单量排名前10省的成功充值金额以及订单充值成功率

统计每小时各产品充值情况，用于了解用户充值习惯，用于上层以该数据为参考预测下一个时间段内的营业额。

### 3.2.4实时统计各省各产品充值概况

帮助上层制定下一步产品倾向战略计划和各省产品战略调整，产品运营方案等。

### 3.2.5每小时的充值笔数和成功笔数和成功的充值金额

通过每小时的充值笔数和成功笔数和成功的充值金额曲线适时调节服务资源降低服务卡顿概率。

### 3.2.6统计每分钟各省的充值情况

统计充值订单的订单量、成功量、总金额、成功金额，该数据用于着重调节某省份的服务器压力。

## 3.3 性能需求分析

由于是模拟系统，机器集群限制了超大数据量的分析、因而只能保证数据拉取速度和数据摄入速度在10MB/S左右，数据处理速度超过10MB/S。要保证平台服务器不会轻易宕机且宕机后可以快速准确的恢复数据处理，从而保证最少量的数据积压。

## 3.4 数据流程分析

数据源为充值系统后端服务日志，使用Flume Exec模式（Shell 命令）监控Source（系统日志文件或文件夹），选用Kafka作为Flume的数据的Sinker，使用内存和磁盘混合方式的传输刘渠道避免数据量过大导致Flume程序占用大量资源，之后数据从Kafka输出到Spark 组件进行微批次处理，最后结果数据流入各数据库，最后被其他项目调用展示。

下图为数据在计算中的流程图。

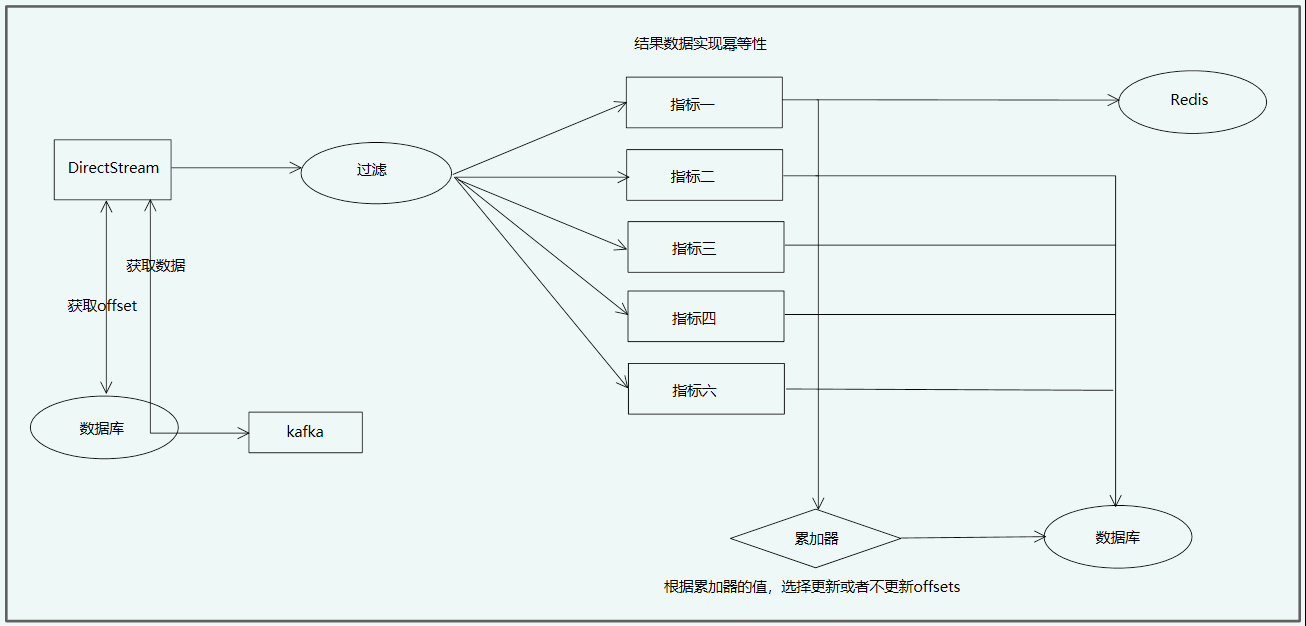


图3-4处理过程数据流程visio

## 3.5 数据库需求分析

根据需求分析，设计数据库表结构如下：

表3-1 Redis事务标记数据字典

|  |
| --- |
| 数据项名称：事务标记  数据项别名：redis\_trans\_key  说明：上次消费过的数据的标记，存放数据处理全局唯一标识  类型：key-value  取值范围及含义：不能为NULL |

表3-2 Redis全网订单数据数据字典

|  |
| --- |
| 数据项名称：全网订单数据  数据项别名：redis\_trans\_value  说明：计算出的数据（订单总量、成功订单量、成功订单金额、订单花费时间）  类型：key-value  取值范围及含义：不能为NULL，以空格为分隔将数据存入Redis |

表3-3 MySQL全网订单唯一id数据字典

|  |
| --- |
| 数据项名称：全网订单数据唯一id  数据项别名：time\_partition\_id  说明：全局id字段  类型：varchar(255)  取值范围及含义：不能为NULL |

表3-4 MySQL全网订单时间标识数据字典

|  |
| --- |
| 数据项名称：全网订单数据产出时间  数据项别名：datehour  说明：标识以小时单位数据的产出时间  类型：varchar(255)  取值范围及含义：不能为NULL |

表3-5 MySQL全网订单成功订单量数据字典

|  |
| --- |
| 数据项名称：全网订单成功订单量  数据项别名：successed\_count  说明：全网订单成功订单量  类型： int  取值范围及含义：不能为NULL |

表3-7 MySQL全网订单成功订单金额数据字典

|  |
| --- |
| 数据项名称：全网订单成功订单金额  数据项别名：chargefee  说明：全网订单成功订单金额  类型：int  取值范围及含义：不能为NULL |

表3-8 MySQL全网订单时间标识数据字典

|  |
| --- |
| 数据项名称：全网订单数据总订单量  数据项别名：order\_count  说明：全网订单数据总订单量  类型：int  取值范围及含义：不能为NULL |

表3-9 MySQL省维度失败订单唯一id数据字典

|  |
| --- |
| 数据项名称：全网订单数据唯一id  数据项别名：time\_partition\_id  说明：全局id字段  类型：varchar(255)  取值范围及含义：不能为NULL |

表3-10 MySQL省维度失败订单省份数据字典

|  |
| --- |
| 数据项名称：订单省份  数据项别名：province  说明：订单省份  类型：varchar(255)  取值范围及含义：不能为NULL |

表3-11 MySQL省维度失败订单订单时间数据字典

|  |
| --- |
| 数据项名称：订单时间  数据项别名：datehour  说明：订单时间  类型：varchar(255)  取值范围及含义：不能为NULL |

表3-12 MySQL元数据TOPIC数据字典

|  |
| --- |
| 数据项名称：元数据主题  数据项别名：topic\_id  说明：元数据主题  类型：varchar(255)  取值范围及含义：不能为NULL |

表3-13 MySQL元数据主题分区号数据字典

|  |
| --- |
| 数据项名称：元数据主题分区  数据项别名：partition\_id  说明：元数据主题分区  类型：int  取值范围及含义：不能为NULL |

表3-14 MySQL元数据主题分区偏移数据字典

|  |
| --- |
| 数据项名称：元数据主题分区偏移  数据项别名：offset\_id  说明：元数据主题分区偏移  类型：bigint  取值范围及含义：不能为NULL |

# 4.平台设计

## 4.1 平台总体结构设计

围绕的是充值系统的日志数据开发本大数据处理平台，作用是在面对大数据量时快速按照维度计算数据，将数据分为时间和地理上的分析得到结果后并存入其他常用数据库，为其他部分分析使用。

记录数据从充值系统产生，使用Flume使用Exec（Shell 命令）方式监控Source（系统日志文件或文件夹），使用Kafka作为上游数据的Sink，使用memory and disk方式的channel避免数据量过大导致Flume程序占用大量资源，将Kafka的Producer ACK副本同步确认机制设置为1或者-1避免出现极端情况下的Kafka数据不一致和数据不完整，之后使用Spark Streaming通过Direct方式连接Kafka获取数据并开始一系列的处理，处理完成之后这里为了保证数据消费的Exactly Once（一次仅以此）语义需要我们的程序管理Kafka数据的Offset（偏移量），这里我们将每个分区的数据更新到MySQL，并且会在程序恢复的时候重新获取MySQL中的Offset重新用以获取数据， 将在几分钟内会进行大量迭代更新的数据存入Redis从而避免使用Spark Streaming窗口函数导致磁盘占用过高。使用的大数据组件都是较为稳定成熟的体系，在之后的平台系统发展中，除非出现平台计算、存储等瓶颈，否则不需要更换组件。

组件架构图如下：

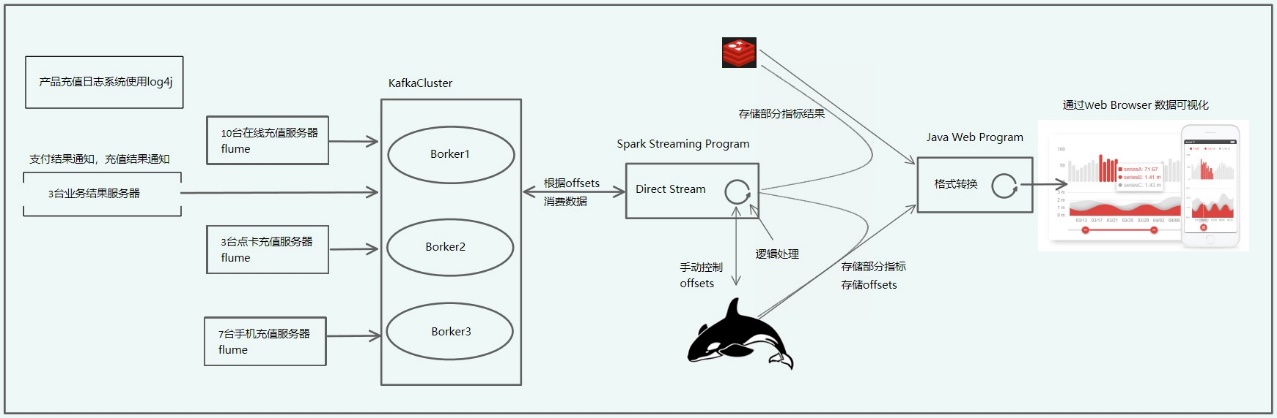


图4-1数据处理平台架构

## 4.2 数据库设计

由于数据库中的数据都是作为最终结果数据，因此数据表之间的关系较为微弱。因此仅对原始数据的映射和结果数据结构进行设计。

表4-1 日志数据字段映射表

|  |  |  |  |
| --- | --- | --- | --- |
| 字段名称 | 字段类型 | 字段是否允许空 | 字段描述 |
| bussinessRst | string | NOT NULL | 结果 |
| channelCode | string | NOT NULL | 渠道 |
| chargefee | string | NOT NULL | 金额 |
| clientIp | string | NOT NULL | Ip |
| endReqTime | string | NOT NULL | 请求结束时间 |
| idType | string | NOT NULL | 请求类型 |
| provinceCode | string | NOT NULL | 请求省份ID |
| requestTime | string | NOT NULL | 请求时间 |

表4-2 offsets元数据表 offsets

|  |  |  |  |
| --- | --- | --- | --- |
| 字段名称 | 字段类型 | 字段是否允许空 | 字段描述 |
| topic\_id | varchar(255) | NOT NULL | Kafka主题名称 |
| partition\_id | int | NOT NULL | Kafka主题分区号 |
| offset\_id | bigint | NOT NULL | 分区消费偏移量 |

表4-3 全网每小时内订单数据表 datehour\_order\_successed\_chargefee

|  |  |  |  |
| --- | --- | --- | --- |
| 字段名称 | 字段类型 | 字段是否允许空 | 字段描述 |
| time\_partition\_id | varchar(255) | NOT NULL | 全局唯一id |
| datehour | varchar(255) | NOT NULL | 时间 |
| successed\_count | int | NOT NULL | 成功订单量 |
| chargefee | int | NOT NULL | 成功订单金额 |
| order\_count | int | NOT NULL | 总订单量 |

表4-4 各省每小时内失败订单数据表 datehour\_province\_failedcount

|  |  |  |  |
| --- | --- | --- | --- |
| 字段名称 | 字段类型 | 字段是否允许空 | 字段描述 |
| time\_partition\_id | varchar(255) | NOT NULL | 全局唯一id |
| datehour | varchar(255) | NOT NULL | 时间 |
| province | varchar(255) | NOT NULL | 省份 |
| failed\_count | int | NOT NULL | 失败订单量 |

表4-5 各省每分钟内订单数据表 datemin\_province\_succount\_ordercount\_succhargefee

|  |  |  |  |
| --- | --- | --- | --- |
| 字段名称 | 字段类型 | 字段是否允许空 | 字段描述 |
| time\_partition\_id | varchar(255) | NOT NULL | 全局唯一id |
| dateminuate | varchar(255) | NOT NULL | 时间 |
| province | varchar(255) | NOT NULL | 省份 |
| order\_count | int | NOT NULL | 总订单量 |
| successed\_count | int | NOT NULL | 成功订单量 |
| chargefee | int | NOT NULL | 成功订单金额 |

表4-6 全网各省总量订单数据表 province\_ordercount\_sucperc\_succharge

|  |  |  |  |
| --- | --- | --- | --- |
| 字段名称 | 字段类型 | 字段是否允许空 | 字段描述 |
| time\_partition\_id | varchar(255) | NOT NULL | 全局唯一id |
| province | varchar(255) | NOT NULL | 时间 |
| order\_count | int | NOT NULL | 总订单量 |
| successed\_perc | Double(11,0) | NOT NULL | 成功订单百分比 |
| chargefee | int | NOT NULL | 成功订单金额 |

# 5.平台实现

## 5.1功能的实现

### 5.1.1 Spark Streaming数据获取的实现

设置Kafka数据流的消费参数时，将Kafka的消费数据offset重置规则设置为“none”以便使用我们写的代码管理offset，使用StreamingContext.getOrCreate方法从检查点中恢复一个StreamingContext或者创建一个新的StreamingContext并创建一个检查点目录，之后将Kafka属性实例、数据流所属的Topic以及刚刚创建的StreamingContext实例传入方法KafkaUtils. createDirectStream中获取抽象数据流（InputDStream）的实例。

在文件中加载城市事实表文件并转换为key-value形式的抽象实例，将该抽象实例使用SparkContext的broadcast方法将其广播到各个Executer中以减少数据处理过程中的网络I/O。

|  |
| --- |
| object AnalyzeContext {  val checkpointPath = "file:///D://test\_Analyze"  val topicSet = Set("test")  val kafkaParams: Map[String, Object] =  Map(elems =  "bootstrap.servers" -> "m1:9092,m2:9092,m3:9092",  "key.deserializer" -> classOf[StringDeserializer],  "value.deserializer" -> classOf[StringDeserializer],  "enable.auto.commit" -> (false: java.lang.Boolean),  "group.id" -> "group02",  "auto.offset.reset" -> "none"  )  def main(args: Array[String]): Unit = {  val conf = new SparkConf()  conf.setMaster("local")  .setAppName("Consume Analyze")  .set("spark.streaming.kafka.maxRatePerPartition", "500000")  .set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")  val sc = new SparkContext(conf)  // 从文件系统恢复或者创建新的数据流上下文  val ssc = StreamingContext.getOrCreate(checkpointPath, () => {  val res = new StreamingContext(conf, Durations.milliseconds(5000))  res.checkpoint(checkpointPath)  res  })  // 获取数据流实例  val ds: InputDStream[ConsumerRecord[String, String]] = KafkaUtils.createDirectStream[String, String]( ssc,  PreferConsistent,  Subscribe[String, String](topicSet, kafkaParams, DataBaseUtils.initOffsets()))  // 加载城市名称和城市id的映射关系  val cityFileInfo: Map[String, String] = sc.textFile("D:\\backup\\workspace\\realTimeAnalyze\\src\\main\\resources\\city.txt")  .map(\_.split(" ")).map(e => (e(0), e(1))).collect().toMap  // 将城市名称和城市id的映射关系广播出去  val cityFileInfoBroadCasted: Broadcast[Map[String, String]] = sc.broadcast(cityFileInfo)  CleanData.cleanData(sc, ds, cityFileInfoBroadCasted)  ssc.start()  ssc.awaitTermination()  }  } |

### 5.1.2 Spark Streaming数据处理功能的实现

根据传入的数据流开始计算各个指标数据；在实现各个数据处理指标之前需要先将数据进行格式转换，使用mapPartitions方法将数据流中的每个String实例转换为Json实例以便于之后的操作，使用filter方法对数据进行过滤仅留下充值请求的日志记录，使得下一步计算的数据都是符合我们需求的数据，由于过滤了不必要的数据因而优化了数据的处理速度，之后将处理后的数据集缓存起来可以供之后的其他的数据计算指标使用。

allOrderChargefeeSuccessCount是关于全网充值信息的元组实例，其被设计用来满足全网统计充值订单总金额和订单量的需求，实现中将部分数据存入了外部内存也就是Redis中，Redis的连接的获取是使用Jedis连接池实现，每个微批次的数据处理时将先从Redis中获取历史数据并将其和本批次数据聚合后的结果再存入Redis达到增量更新的效果。

provinceDatehourFailCount是关于各省份充值失败数据的元组实例，其被设计用来满足每小时各省份充值失败数据量的需求。因为该需求是计算每个小时的失败数据，所以将数据元组中添加一条最细粒度Floor到小时的时间信息（如：2019-01-01 08:30:30被处理为2019-01-01 08），得到每条数据的时间后进行有状态计算便需要将一个小时内的数据checkpoint到文件系统中，遇到下一批数据时使用provinceDatehourFailAggFunction函数对数据进行增量计算。最终结果实例为provinceDatehourFailCountResult。

最后将结果数据事务性地存储到MySQL数据库中。

|  |
| --- |
| object CleanData {  val checkpointTimeMinuate: Int = 60 \* 1000  private val checkpointTimeMillis: Int = 10 \* 1000  private val serviceName: String = "sendRechargeReq".toLowerCase()  private val serviceNameKey: String = "serviceName".toLowerCase  private val bussinessRstKey: String = "bussinessRst".toLowerCase()  private val provinceCodeKey: String = "provinceCode".toLowerCase()  private val chargefeeKey: String = "chargefee".toLowerCase()  private val startReqTimeKey: String = "startReqTime".toLowerCase()  private val endReqTimeKey: String = "endReqTime".toLowerCase()  private val successCode: String = "0000"  private val transAcc: LongAccumulator = new LongAccumulator()  def cleanData(sc: SparkContext, ds: InputDStream[ConsumerRecord[String  , String]], cityFileInfoBroadCasted: Broadcast[Map[String,  String]]): Unit = {  sc.register(transAcc, "Transaction Accumulator")  transAcc.reset()  sc.setLogLevel("ERROR")  val lines: DStream[(JSONObject, ConsumerRecord[String, String])]  = ds.mapPartitions(it => {  it.map(elem => {  val jsonObj: JSONObject = JSON.parseObject(elem.value().toLowerCase)  (jsonObj, elem)  })  })  val reChargeNotifyReqLines = lines.filter(\_.\_1.getString(serviceNameKey)  == serviceName)  reChargeNotifyReqLines.cache()  val allOrderChargefeeSuccessCount =  reChargeNotifyReqLines.mapPartitions(it => {  it.map(elem => {  val jsonObj = elem.\_1  val successedCount: Int = if (jsonObj.get(bussinessRstKey)  .toString == successCode) 1 else 0  val chargefee:Long = if (successedCount == 1)  jsonObj.getLongValue(chargefeeKey) else 0l  val startTime = jsonObj.get(startReqTimeKey).toString  val endTime = jsonObj.get(endReqTimeKey).toString  val subTimeMillis = TimeUtiles.subMillis(startTime, endTime)  (1, (successedCount, chargefee, subTimeMillis))  })  }).reduce((a, b) => (a.\_1 + b.\_1, (a.\_2.\_1 + b.\_2.\_1,  a.\_2.\_2 + b.\_2.\_2, a.\_2.\_3 + b.\_2.\_3)))  .map(e => (0, e.\_1, e.\_2.\_1, e.\_2.\_2, e.\_2.\_3))  allOrderChargefeeSuccessCount  .map(elem => (elem.\_1, elem.\_2, elem.\_3, elem.\_4, elem.\_5.toDouble / elem.\_2))  .foreachRDD((rdd, time) => {  rdd.foreachPartition(it => {  val uniqueId = generateUniqueId(time, TaskContext.get.partitionId())  val jedisConn = JedisConnectionPool.getConnection()  it.foreach(elem => {  //本批次计算的数据  var allCount: Int = elem.\_2  var successedCount: Int = elem.\_3  var chargefee: Long = elem.\_4  var ageTimeMillis: Double = elem.\_5  //获取历史数据  val key = jedisConn.get("redis")  if (key != null && !key.isEmpty && jedisConn.get(key)!=null &&  jedisConn.get(key).split(" ").length>=4) {  val valuesJ = jedisConn.get(key).split(" ")  val allCountJ = valuesJ(0).toInt  val successedCountJ = valuesJ(1).toInt  val chargefeeJ = valuesJ(2).toLong  val ageTimeMillisJ = valuesJ(3).toDouble  allCount += allCountJ  successedCount += successedCountJ  chargefee += chargefeeJ  ageTimeMillis += ageTimeMillisJ  }  // 更新数据  val elemString: String = s"$allCount $successedCount  $chargefee $ageTimeMillis"  jedisConn.set(s"$uniqueId", elemString)  // 将key的更新放到DbUtils里边保证更新！！！  })  jedisConn.close()  })  })  /\*\* 指标二  \*  \* 要求：每小时各个省份的充值失败数据量  \*/  val provinceDatehourFailAggFunction = (key: String, count:  Option[Int], state: State[Int]) => {  (key, count.getOrElse(0) + state.getOption().getOrElse(0))  }  val provinceDatehourFailCount = reChargeNotifyReqLines  .mapPartitions(it => {  it.map(elem => {  val jsonObj = elem.\_1  val provinceCode = jsonObj.get(provinceCodeKey).toString  val failedCount: Int = if (jsonObj.get(bussinessRstKey)  .toString == successCode) 0 else 1  val time = jsonObj.get(startReqTimeKey).toString  val datehour = TimeUtiles.formatDateHour(time)  val dateHourProvince = s"$datehour:$provinceCode"  (dateHourProvince, failedCount)  }).filter(\_.\_2 != 0)  }).reduceByKey(\_ + \_)  provinceDatehourFailCount.checkpoint(Durations.milliseconds(checkpointTimeMillis))  val provinceDatehourFailCountResult = provinceDatehourFailCount  .mapWithState(StateSpec.function(provinceDatehourFailAggFunction))  .map(e => {  val strs = e.\_1.split(":")  val province = cityFileInfoBroadCasted.value.getOrElse(strs(1), strs(1))  val datehour = strs(0)  (s"$datehour:$province", e.\_2)  })  provinceDatehourFailCountResult.foreachRDD((rdd, time) => {  rdd.foreachPartition(it => {  val list = new ListBuffer[(String, (String, Int))]  val id = generateUniqueId(time, TaskContext.getPartitionId())  it.foreach(elem => {  list.append((id, elem))  })  val state = DataBaseUtils.replaceProvinceDatehourFailCountBatch(list.toList)  transAcc.add(state)  })  })  /\*\* 指标三  \* 要求：订单量排名前10的省，  \* 统计每个省份的订单成功率（只保留一位小数），成功的充值金额  \*/  // todo 写入数据库 Mysql  val provinceOrderSuccessCountAggStateFunc = (key: String, count:  Option[(Int, Int, Int)], state: State[(Int, Int, Int)]) => {  val c = count.getOrElse((0, 0, 0))  val s = state.getOption().getOrElse((0, 0, 0))  (key, c.\_1 + s.\_1, c.\_2 + s.\_2, c.\_3 + s.\_3)  }  val provinceOrderSuccessCount = reChargeNotifyReqLines  .mapPartitions(it => {  it.map(elem => {  val jsonObj = elem.\_1  val provinceCode = jsonObj.get(provinceCodeKey).toString  val successedCount: Int = if (jsonObj.get(bussinessRstKey)  .toString == successCode) 1 else 0  val chargefee: Int = if (successedCount == 1) jsonObj.getIntValue(chargefeeKey) else 0  (provinceCode, (1, successedCount, chargefee))  })  }).reduceByKey((a, b) => (a.\_1 + b.\_1, a.\_2 + b.\_2, a.\_3 + b.\_3))  provinceOrderSuccessCount.checkpoint(Durations.milliseconds(checkpointTimeMillis))  val provinceOrderSuccessCountInitValue = sc.makeRDD(Array(("", (0, 0, 0))))  val provinceOrderSuccessCountAgged = provinceOrderSuccessCount  .mapWithState(StateSpec.function(provinceOrderSuccessCountAggStateFunc)  .initialState(provinceOrderSuccessCountInitValue))  val provinceOrderSuccessCountAggedBatchResult = provinceOrderSuccessCountAgged  .map(elem => (elem.\_1, elem.\_2, elem.\_3.toDouble / elem.\_2, elem.\_4))  .transform(rdd => rdd.sortBy(\_.\_2, ascending = false))  val provinceOrderSuccessCountAggedResult = provinceOrderSuccessCountAggedBatchResult  .mapPartitions(it => {  val cityMap: Map[String, String] = cityFileInfoBroadCasted.value  it.map(elem => {  val cityName = cityMap.getOrElse(elem.\_1, elem.\_1)  (cityName, elem.\_2, elem.\_3, elem.\_4)  })  })  provinceOrderSuccessCountAggedResult.map(("指标三", \_)).print(10)  // 写出到数据库  provinceOrderSuccessCountAggedResult.foreachRDD((rdd, time) => {  rdd.foreachPartition(it => {  val id = generateUniqueId(time, TaskContext.getPartitionId())  val list = new ListBuffer[(String, (String, Int, Double, Int))]  it.foreach(elem => {  list.append((id, elem))  })  val state = DataBaseUtils.replaceProvinceOrderSuccessCount(list.toList)  transAcc.add(state)  })  })  /\*\* 指标四  \* 要求:每小时的笔数、成功数以及成功的金额  \*/  // todo 写出到数据库  val datehourAggFuncation = (key: String, count: Option[(Int, Int, Int)]  , state: State[(Int, Int, Int)]) => {  val c = count.getOrElse((0, 0, 0))  val s = state.getOption().getOrElse((0, 0, 0))  (key, (c.\_1 + s.\_1, c.\_2 + s.\_2, c.\_3 + s.\_3))  }  val datehourOrderSuccessChargefee = reChargeNotifyReqLines  .mapPartitions(it => {  it.map(elem => {  val jsonObj = elem.\_1  val successedCount: Int = if (jsonObj.get(bussinessRstKey)  .toString == successCode) 1 else 0  val chargefee: Int = if (successedCount == 1) jsonObj.getIntValue(chargefeeKey) else 0  val startTime = jsonObj.get(startReqTimeKey).toString  val datehour = TimeUtiles.formatDateHour(startTime)  //(2017/04/12/03,(23,12003,23))  (datehour, (successedCount, chargefee, 1))  })  }).reduceByKey((a, b) => (a.\_1 + b.\_1, b.\_1 + b.\_2, a.\_3 + b.\_3))  datehourOrderSuccessChargefee.checkpoint(Durations.milliseconds(checkpointTimeMillis))  val datehourOrderSuccessChargefeeResult = datehourOrderSuccessChargefee  .mapWithState(StateSpec.function(datehourAggFuncation))  //写出到数据库  datehourOrderSuccessChargefeeResult.foreachRDD((rdd, time) => {  rdd.foreachPartition(it => {  val id = generateUniqueId(time, TaskContext.getPartitionId())  val list = new ListBuffer[(String, (String, (Int, Int, Int)))]  it.foreach(elem => {  list.append((id, elem))  })  val state = DataBaseUtils.replaceDatehourOrderSuccessChargefee(list.toList)  transAcc.add(state)  })  })  datehourOrderSuccessChargefeeResult.map(("指标四", \_)).print(10)  /\*\* 指标六  \* 要求：以省份为维度 ,统计每分钟各省的充值笔数和充值金额  \*/  //有状态聚合函数  val provinceDHMWAggFuncation = (key: String, count: Option[(Int, Int, Int)]  , state: State[(Int, Int, Int)]) => {  val c = count.getOrElse((0, 0, 0))  val s = state.getOption().getOrElse((0, 0, 0))  (key, (c.\_1 + s.\_1, c.\_2 + s.\_2, c.\_3 + s.\_3))  }  val provinceDateHMWin: DStream[(String, (Int, Int, Int))] =  reChargeNotifyReqLines  .mapPartitions(it => {  it.map(elem => {  val jsonObj = elem.\_1  val successedCount: Int = if (jsonObj.get(bussinessRstKey)  .toString == successCode) 1 else 0  val chargefee: Int=if (successedCount == 1) jsonObj.getIntValue(chargefeeKey) else 0  val startTime = jsonObj.get(startReqTimeKey).toString  val dateHourMin = TimeUtiles.formatDateMinuate(startTime)  val provinceCode = jsonObj.get(provinceCodeKey).toString  (s"$dateHourMin:$provinceCode", (1, successedCount, chargefee))  })  }).reduceByKey((a, b) => (a.\_1 + b.\_1, a.\_2 + b.\_2, a.\_3 + b.\_3))  .reduceByKeyAndWindow((a: (Int, Int, Int), b: (Int, Int, Int)) =>  (a.\_1 + b.\_1, a.\_2 + b.\_2, a.\_3 + b.\_3)  , Durations.milliseconds(60 \* 1000), Durations.milliseconds(60 \* 1000))  //有状态聚合  provinceDateHMWin.checkpoint(Durations.milliseconds(checkpointTimeMinuate))  val provinceDateHMResult: DStream[(String, (Int, Int, Int))] = provinceDateHMWin  .mapWithState(StateSpec.function(provinceDHMWAggFuncation))  .map(elem => {  val provinceCodeDateHM = elem.\_1.split(":")  val cityName: Map[String, String] = cityFileInfoBroadCasted.value  val province = cityName.getOrElse(provinceCodeDateHM(1), provinceCodeDateHM(1))  val DateHM = provinceCodeDateHM(0)  (s"$DateHM:$province", elem.\_2)  })  //写出到数据库  provinceDateHMResult.foreachRDD((rdd, time) => {  rdd.foreachPartition(it => {  val list = new ListBuffer[(String, (String, (Int, Int, Int)))]  val id = generateUniqueId(time, TaskContext.get.partitionId())  it.foreach(elem => {  list.append((id, elem))  })  val state: Int = DataBaseUtils.replaceProvinceDateHM(list.toList)  transAcc.add(state)  })  })  provinceDateHMResult.map(("指标六", \_)).print(10)  //保存offsets  ds.foreachRDD((rdd, time) => {  if (transAcc.value == 0) {  rdd.foreachPartition(it => {  val uniqueId = generateUniqueId(time, TaskContext.getPartitionId())  val list = new ListBuffer[(TopicPartition, Long)]  it.foreach(elem => {  val tp: TopicPartition = new TopicPartition(elem.topic(), elem.partition())  list.append((tp, elem.offset()))  })  DataBaseUtils.updateOffset(uniqueId, list.toList)  })  }  transAcc.reset()  //手动提交offset  val offsetRanges: Array[OffsetRange] = rdd.asInstanceOf[HasOffsetRanges].offsetRanges  val offsetsFromSQL: Map[TopicPartition, Long] = DataBaseUtils.initOffsets()  val offsetRangeSyncSQL: Array[OffsetRange] = offsetRanges.map(offsetRange=>{  val offsetFromSQL: Long = offsetsFromSQL.getOrElse(offsetRange.topicPartition(),  offsetRange.untilOffset)  OffsetRange(offsetRange.topicPartition().topic(), offsetRange.topicPartition().partition(),  offsetRange.fromOffset, offsetFromSQL)  })  ds.asInstanceOf[CanCommitOffsets].commitAsync(offsetRangeSyncSQL)  })  }  def generateUniqueId(time: Time, partitionId: Int): String = {  time.milliseconds + " " + partitionId  }  def update1Key(uniqueId: String): Int = {  val jedisConn = JedisConnectionPool.getConnection()  jedisConn.set("redis", s"$uniqueId")  //验证更新  if (jedisConn.get("redis") == s"$uniqueId") {  0  } else {  1  }  }  } |

# 6.平台测试

### **6.1 测试的目的**

给出特定的一部分数据，将其输入到系统中，测试计算的准确性、数据的完整性、数据的一致性，确保数据符合我们的需求。

### **6.2 测试的内容**

数据格式：{"sysId":"15","startRequestTime":"2019-04-12080615365", "bussinessRequest":"0000","requestId":"201904125059946231037","channelCode":"0000","chargefee":"1111", ,"endRequestTime":"20191212235545999","idType":"01","interFacRequest":"0000","orderId":"20202020202020202020","provinceCode":"931","requestMsg":"成功","shouldfee":"2985","serverIp":"188.154.89.58", "serverPort":"80000","logoutTime":"2019041211"}

### **6.3 测试的结果**

需求一运行结果如图6.1所示，2017年4月12日03时成功充值订单单数为462单，成功总金额为10001元，总订单量为462单。



图6-1 datehour\_order\_successed\_chargefee表中的结果数据示例图

需求二运行结果如图6.2所示，广东省份2017年4月12日03时失败订单为16单。



图6-2 datehour\_province\_failedcount表中的结果数据示例图

需求三运行结果如图6.3所示，内蒙古省份2017年4月12日06时23分订单总量为1单成功，订单总量为1单，成功订单金额为5250元。



图6-3 datemin\_province\_succount\_ordercount\_succhargefee表中的结果数据示例图

数据的消费记录如图6.4所示，test主题下，0号分区，数据消费偏移量为9248。表示已经从0条数据开始消费了9248条数据。



图6-4 offsets表中的结果数据示例图

需求四运行结果如图6.5所示，上海市总订单数为3单，成功订单比例为100%，订单金额为66000元。



图6-5 province\_ordercount\_sucperc\_succharge表中的结果数据示例图

经过测试结果与模拟数据聚合结果对比，测试结果符合我我们对数据准确性的要求，测试阶段完成。

# 7.总结与展望

## 7.1总结

我们实时大数据处理平台实现了按省份和时间维度的数据的统计计算，中间因为各种原因而改变了系统的数据存储介质，如将中间结果数据存入Redis，最终经过多次的修改和系统测试后，项目终于在保证数据计算准确性的情况下满足了我们的基础需求。

## 7.2展望

通过这个项目的开发，使得我更加熟悉了数据处理的过程中数据是如何分发计算的，一个项目的开发流程、数据结构的设计、数据计算的方式等都有了更深的理解。深刻的理解了大量的开发工作不是一个人能够完成的，为了保证开发效率和开发质量必须要学会团队合作。

本项目最终的开发成功，为我们下一次的开发积累了非常宝贵的经验。

# 参考文献

1. Tom White. Hadoop权威指南[M]. 清华大学出版社，2017.
2. Neha Narkhede等. [Kafka权威指南](http://ss.zhizhen.com/detail_38502727e7500f26b74322febee538aca4db398324208f491921b0a3ea25510134114c969f2eae5c737ab1a312d53c2188c6ea0bd81c1a7673fb553e989be0c5a6c2f4f2a575d8b3015579cd82a9a125?&apistrclassfy=0_18_17)[M]. 人民邮电出版社，2018.
3. Lars George等. HBase权威指南[M]. 人民邮电出版社，2013.
4. 施瓦茨，扎伊采夫，特卡琴科. 高性能MySQL[M]. 电子工业出版社，2013.
5. Holden Karau等. Spark快速大数据分析[M]. 人民邮电出版社，2015.
6. Sandy Ryza等. Spark高级数据分析[M]. 人民邮电出版社，2018.
7. 布鲁斯·埃克尔等. Scala编程思想[M]. 机械工业出版社，2015.
8. Tyler Akidau等.Streaming Systems[M]. O'Reilly Media出版，2018.
9. 孔祥盛. [MySQL核心技术与最佳实践](http://210.44.126.8:8080/opac/item.php?marc_no=0000359280)[M]. 人民邮电出版社，2014.
10. Charles Bell，Mats Kindahl，Lars Thalmann著. 高可用MySQL[M]. 电子工业出版社，2015
11. Sveta Smirnova著. MySQL排错指南[M]. 人民邮电出版社，2015
12. Michael Kofler著. MySQL 5权威指南[M]. 人民邮电出版社，2006
13. 冯登国, 张敏, 李昊. 大数据安全与隐私保护[J]. 计算机学报, 2014, 37(1):246-258.

# 致 谢

首先感谢我们聊城大学的各位教师的知识上的谆谆教诲、以及讲师在我论文课题上的引导，没有老师的教诲和四年的积淀，这个项目是无法在规定时间内完成的，聊大部分的改变了我对世界、社会、学习等事看法，也同时改变了我做事的风格，我学会了看问题需要忽略表象看本质、解决问题则需要事无巨细注重细节，这四年的大学生活将是我以后人生的一大财富。

大学四年的学习生活如白驹过隙，有时候会想好像自己还是一名刚入校的大学生，在大学里，我真的成长了许多，能清楚的看到自己身上的变化。

在此，我要对我的毕业设计指导老师韩玉艳老师表示衷心的感谢，感谢老师的鼓励和支持，衷心地祝福韩老师工作顺利、身体健康，聊大能够培育出更多的优秀的人才。

另外，我还要感谢我的父母，无论我在哪里，他们一直都是我的后盾，无论我选择什么样的事情，他们总会相信我、支持我。

最后，再一次向这个过程中帮助我的讲师、同学们表示衷心的感谢。