如何基于 Spark Streaming 构建实时计算平台

前言

随着互联网技术的迅速发展，用户对于数据处理的时效性、准确性与稳定性要求越来越高，如何构建一个稳定易用并提供齐备的监控与预警功能的实时计算平台也成了很多公司一个很大的挑战。

自2015年携程实时计算平台搭建以来，经过两年多不断的技术演进，目前实时集群规模已达上百台，平台涵盖各个SBU与公共部门数百个实时应用，全年JStorm集群稳定性达到100%。目前实时平台主要基于JStorm与Spark Streaming构建而成，相信关注携程实时平台的朋友在去年已经看到一篇关于携程实时平台的分享：携程实时大数据平台实践分享。

本次分享将着重于介绍携程如何基于Spark Streaming构建实时计算平台，文章将从以下几个方面分别阐述平台的构建与应用：

- Spark Streaming vs JStorm
- Spark Streaming设计与封装
- Spark Streaming在携程的实践
- 曾经踩过的坑
- 未来展望

Spark Streaming vs JStorm

携程实时平台在接入Spark Streaming之前，JStorm已稳定运行有一年半，基本能够满足大部分的应用场景。接入Spark Streaming主要有以下几点考虑：首先携程使用的JStorm版本为2.1.1版本，此版本的JStorm封装与抽象程度较低，并没有提供High Level抽象方法以及对窗口、状态和Sql等方面的功能支持，这大大的提高了用户使用JStorm实现实时应用的门槛以及开发复杂实时应用场景的难度。在这几个方面，Spark Streaming表现就相对好的多，不但提供了高度集成的抽象方法（各种算子），并且用户还可以与Spark SQL相结合直接使用SQL处理数据。

其次，用户在处理数据的过程中往往需要维护两套数据处理逻辑，实时计算使用JStorm，离线计算使用Hive或Spark。为了降低开发和维护成本，实现流式与离线计算引擎的统一，Spark为此提供了良好的支撑。

最后，在引入Spark Streaming之前，我们重点分析了Spark与Flink两套技术的引入成本。Flink当时的版本为1.2版本，Spark的版本为2.0.1。相比较于Spark，Flink在SQL与MLlib上的支持相对弱于Spark，并且公司许多部门都是基于Spark SQL与MLlib开发离线任务与算法模型，使得大大降低了用户使用Spark的学习成本。

下图简单的给出了当前我们使用Spark Streaming与JStorm的对比：

	Spark Streaming	JStorm
Guarantee	Exactly Once	At Least Once
Latency	High	Very Low
Throughput	High	Low
Processing Model	Micro-batch	Streaming
Stateful Operation	Yes	No
Window Operation	Yes	No
SQL Support	Yes	No
Levels of Abstraction	High Level	Low Level

Spark Streaming设计与封装

在接入Spark Streaming的初期，首先需要考虑的是如何基于现有的实时平台无缝的嵌入Spark Streaming。原先的实时平台已经包含了许多功能：元数据管理、监控与告警等功能，所以第一步我们先针对Spark Streaming进行了封装并提供了丰富的功能。整套体系总共包含了Muise Spark Core、 Muise Portal以及外部系统。

Muise Spark Core

Muise Spark Core是我们基于Spark Streaming实现的二次封装，用于支持携程多种消息队列，其中Hermes Kafka与源生的Kafka基于Direct Approach的方式消费数据，Hermes Mysql与Qmq基于Receiver的方式消费数据。接下来将要讲的诸多特性主要是针对Kafka类型的数据源。

Muise spark core主要包含了以下特性：

- Kafka Offset自动管理
- 支持Exactly Once与At Least Once语义
- 提供Metric注册系统，用户可注册自定义metric
- 基于系统与用户自定义metric进行预警
- Long running on Yarn，提供容错机制

Kafka Offset自动管理

封装muise spark core的第一目标就是简单易用，让用户以最简单的方式能够上手使用Spark Streaming。首先我们实现了帮助用户自动读取与存储Kafka Offset的功能，用户无需关心Offset是如何被处理的。其次我们也对Kafka Offset的有效性进行了校验，有的用户的作业可能在停止了较长时间后重新运行会出现Offset失效的情形，我们也对此作了对应的操作，目前的操作是将失效的Offset设置为当前有效的最老的Offset。下图展现了用户基于muise spark core编写一个Spark streaming作业的简单示例，用户只需要短短几行代码即可完成代码的初始化并创建好对应的DStream：


```
val topic = "ubt.custom";
val appName = "ubt.custom.example"

val sparkConf = MuiseSparkConfig.getDefaultSparkConfig.setAppName(appName)
                .setMaster("local[*]")   //仅在本地测试时配置，通过portal提交运行时需删除
val ssc = new StreamingContext(sparkConf, Seconds(5))

val km = new CtripKafkaManager(appName, topic)
km.setSkipSavedOffset(true)

val dstream = km.createDirectStream[CustomEvent](ssc)
```

默认情况下，作业每次都是基于上次存储的Kafka Offset继续消费，但是用户也可以自行决定Offset的消费起点。下图中展示了设置消费起点的三种方式:

```
//创建kafkaManager
val km = new CtripKafkaManager(appName,topic);
km.setSkipSavedOffset(true);  //是否跳过存储的offset，使用新的offset开始消费
km.setUseLatestOffset(true); //从最新的offset开始读，还是从最老的offset开始读
km.setStartOffsetTime(System.currentTimeMillis() - 60*60*1000); //读取指定时间起始的offset,
```

Exactly Once的实现

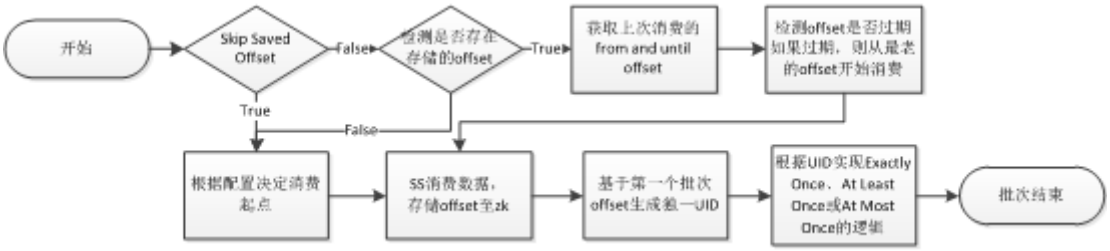
如果实时作业要实现端对端的exactly once则需要数据源、数据处理与数据存储的三个阶段都保证exactly once的语义。目前基于Kafka Direct API加上Spark RDD算子精确一次的保证能够实现端对端的exactly once的语义。在数据存储阶段一般实现exactly once需要保证存储的过程是幂等操作或事务操作。很多系统本身就支持了幂等操作，比如相同数据写hdfs同一个文件，这本身就是幂等操作，保证了多次操作最终获取的值还是相同；HBase、ElasticSearch与redis等都能够实现幂等操作。对于关系型数据库的操作一般都是能够支持事务性操作。

官方在创建DirectKafkaInputStream时只需要输入消费Kafka的From Offset，然后其自行获取本次消费的End Offset，也就是当前最新的Offset。保存的Offset是本批次的End Offset，下次消费从上次的End Offset开始消费。当程序宕机或重启任务后，这其中存在一些问题。如果在数据处理完成前存储Offset，则可能存在作业处理数据失败与作业宕机等情况，重启后会无法追溯上次处理的数据导致数据出现丢失。如果在数据处理完成后存储Offset，但是存储Offset过程中发生失败或作业宕机等情况，则在重启后会重复消费上次已经消费过的数据。而且此时又无法保证重启后消费的数据与宕机前的数据量相同数据相当，这又会引入另外一个问题，如果是基于聚合统计指标作更新操作，这会带来无法判断上次数据是否已经更新成功。

所以在muise spark core中我们加入了自己的实现用以保证Exactly once的语义。具体的实现是我们对Spark源码进行了改造，保证在创建DirectKafkaInputStream可以同时输入From Offset与End Offset，并且我们在存储Kafka Offset的时候保存了每个批次的起始Offset与结束Offset，具体格式如下：


```
{"partitions":[{"partition":0,"untilOffset":13150695,"fromOffset":13150680},
{"partition":2,"untilOffset":20492950,"fromOffset":20492950},
{"partition":1,"untilOffset":42949406,"fromOffset":42949406},
{"partition":3,"untilOffset":13186638,"fromOffset":13186638},
{"partition":4,"untilOffset":17024073,"fromOffset":17024073}],
"topic":"ubt.custom"}
```

如此做的用意在于能够确保无论是宕机还是人为重启，重启后的第一个批次与重启前的最后一个批次数据一模一样。这样的设计使得后面用户在后面对于第一个批次的数据处理非常灵活可变，如果用户直接忽略第一个批次的数据，那此时保证的是at most once的语义，因为我们无法获知重启前的最后一个批次数据操作是否有成功完成；如果用户依照原有逻辑处理第一个批次的数据，不对其做去重操作，那此时保证的是at least once的语义，最终结果中可能存在重复数据；最后如果用户想要实现exactly once，muise spark core提供了根据topic、partition与offset生成UID的功能，只要确保两个批次消费的Offset相同，则最终生成的UID也相同，用户可以根据此UID作为判断上个批次数据是否有存储成功的依据。下面简单的给出了重启后第一个批次操作的行为。



Metrics系统

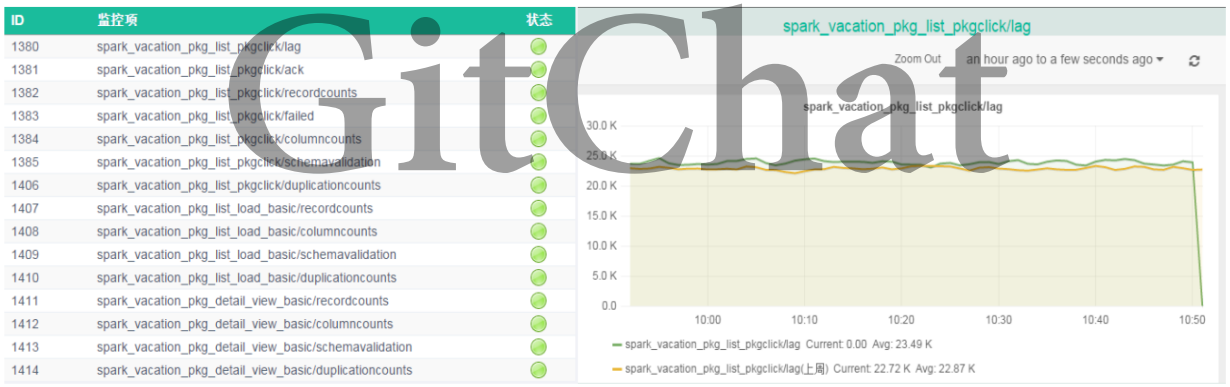
Musie spark core基于Spark本身的metrics系统进行了改造，添加了许多定制的metrics，并且向用户暴露了metrics注册接口，用户可以非常方便的注册自己的metrics并在程序中更新metrics的数值。最后所有的metrics会根据作业设定的批次间隔写入Graphite，基于公司定制的预警系统进行报警，前端可以通过Grafana展现各项metrics指标。

Muise spark core本身定制的metrics包含以下三种：

- Fail 批次时间内spark task失败次数超过4次便报警，用于监控程序的运行状态。
- Ack 批次时间内spark streaming处理的数据量小0便报警，用于监控程序是否在正常消费数据。
- Lag 批次时间内数据消费延迟大于设定值便报警。

其中由于我们大部分作业开启了Back Pressure功能，这就导致在Spark UI中看到每个批次数据都能在正常时间内消费完成，然而可能此时kafka中已经积压了大量数据，故每个批次我们都会计算当前消费时间与数据本身时间的一个平均差值，如果这个差值大于批次时间，说明本身数据消费就已经存在了延迟。

下图展现了预警系统中，基于用户自定义注册的Metrics以及系统定制的Metrics进行预警。



容错

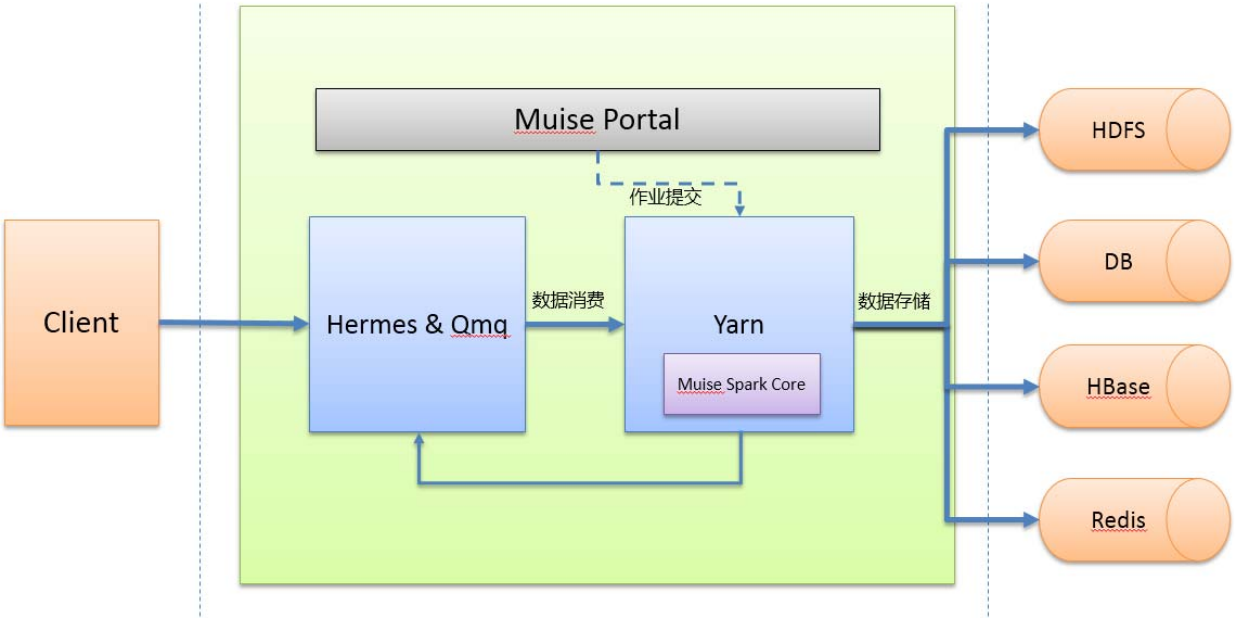
其实在上面Exactly Once一章中已经详细的描述了muise spark core如何在程序宕机后能够保证数据正确的处理。但是为了能够让Spark Sreaming能够长时间稳定的运行在Yarn集群上，还需要添加许多配置，感兴趣的朋友可以查看：Long running Spark Streaming Jobs on Yarn Cluster。

除了上述容错保证之外，Muise Portal（后面会讲）也提供了对Spark Streaming作业定时检测的功能。目前每过5分钟对当前所有数据库中状态标记为Running的Spark Streaming作业进行状态检测，通过Yarn提供的REST APIs可以根据每个作业的Application Id查询作业在Yarn上的状态，如果状态处于非运行状态，则会尝试重启作业。

Muise Portal

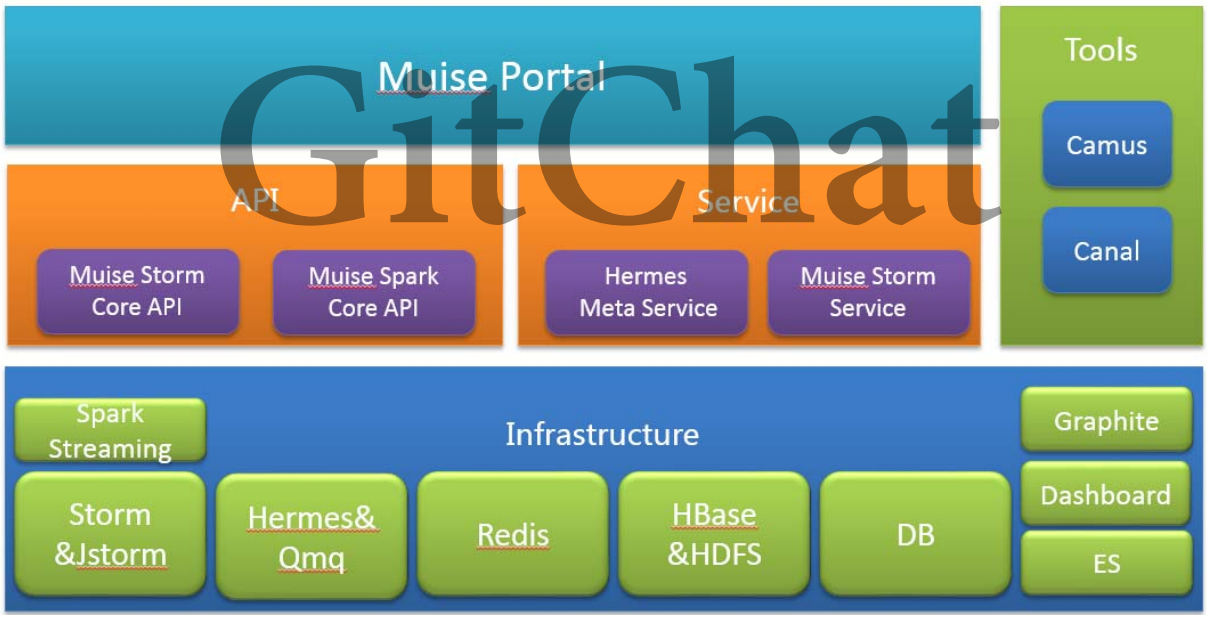
在封装完所有的Spark Streaming之后，我们就需要有一个平台能够管理配置作业，Muise Portal就是这样的存在。Muise Portal目前主要支持了Storm与Spark Streaming两类作业，支持新建作业、Jar包发布、作业运行与停止等一系列功能。下图展现了新建作业的界面：

Spark Streaming作业基于Yarn Cluster模式运行，所有作业通过在Muise Portal上的Spark客户端提交到Yarn集群上运行。具体的一个作业运行流程如下图所示：



整体架构

最后这边给出一下目前携程实时平台的整体架构。



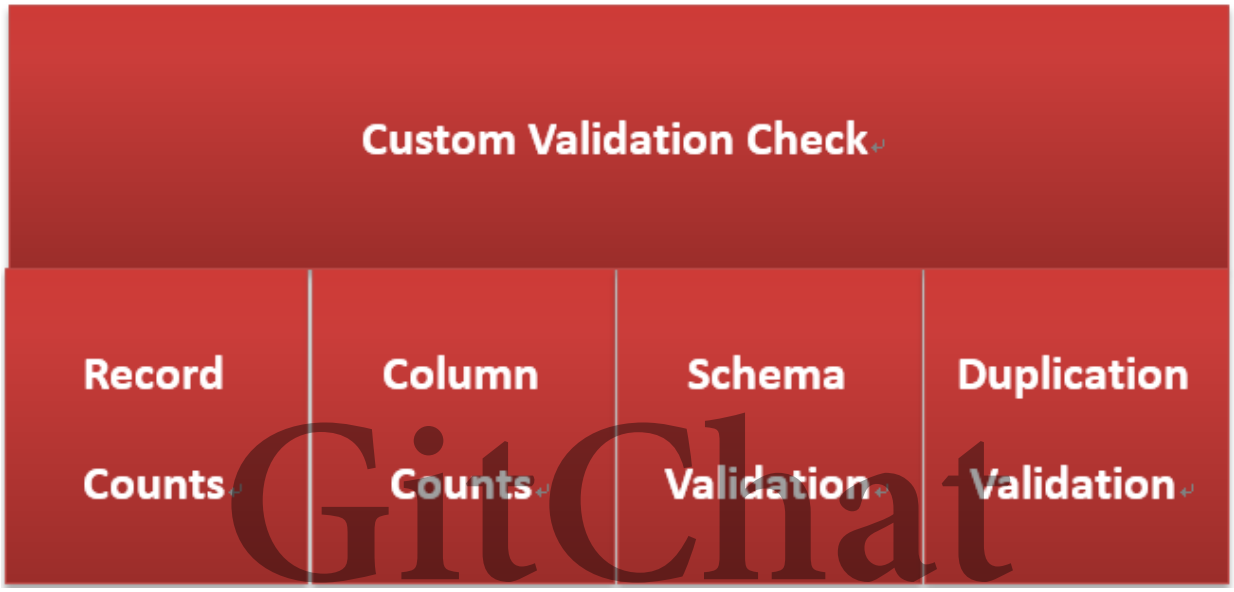
Spark Streaming在携程的实践

目前Spark Streaming在携程的业务场景主要可以分为以下几块：ETL、实时报表统计、个性化推荐类的营销场景以及风控与安全的应用。从抽象上来说，主要可以分为数据过滤抽取、数据指标统计与模型算法的使用。

ETL

如今市面上有形形色色的工具可以从Kafka实时消费数据并进行过滤清洗最终落地到对应的存储系统，如：Camus、Flume等。相比较于此类产品，Spark Streaming的优势首先在于可以支持更为复杂的处理逻辑，其次基于Yarn系统的资源调度使得Spark Streaming的资源配置更加灵活，最后用户可以将Spark RDD数据转换成Spark Dataframe数据，使得可以与Spark SQL相结合，并且最终将数据输出到HDFS和Alluxio等分布式文件系统时可以存储为Parquet之类的格式化数据，用户在后续使用Spark SQL处理数据时更为的简便。

目前在ETL使用场景中较为典型的是携程度假部门的Data Lake应用，度假部门使用Spark Streaming对数据做ETL操作最终将数据存储至Alluxio，期间基于muise-spark-core的自定义metric功能对数据的数据量、字段数、数据格式与重复数据进行了数据质量校验与监控，具体的监控预警已在上面说过。



实时报表统计

实时报表统计与展现也是Spark Streaming使用较多的一个场景，数据可以基于Process Time统计，也可以基于Event Time统计。由于本身Spark Streaming不同批次的job可以视为一个个的滚动窗口，某个独立的窗口中包含了多个时间段的数据，这使得使用Spark Streaming基于Event Time统计时存在一定的限制。一般较为常用的方式是统计每个批次中不同时间维度的累积值并导入到外部系统，如ES；然后在报表展现的时基于时间做二次聚合获得完整的累加值最终求得聚合值。下图展示了携程IBU基于Spark Streaming实现的实时看板。

个性化推荐与风控安全

这两类应用的共同点莫过于它们都需要基于算法模型对用户的行为作出相对应的预测或分类，携程目前所有模型都是基于离线数据每天定时离线训练。在引入Spark Streaming之后，许多部门开始积极的尝试特征的实时提取、模型的在线训练。并且Spark Streaming可以很好的与Spark MLlib相结合，其中最为成功的案例为信安部门以前是基于各类过滤条件抓取攻击请求，后来他们采用离线模型训练，Spark Streaming加Spark MLlib对用户进行实时预测，性能上较JStorm（基于大量正则表达式匹配用户，十分消耗CPU）提高了十倍，漏报率降低了20%。

曾经踩过的坑

目前携程的Spark Streaming作业运行的YARN集群与离线作业同属一个集群，这对作业无论是性能还是稳定性都带来了诸多影响。尤其是当YARN或者Hadoop集群需要更新维护重启服务时，在很大程度上会导致Spark Streaming作业出现报错、挂掉等状况，虽然有诸多的容错保障，但也会导致数据积压数据处理延迟。后期将会独立部署Hadoop与Yarn集群，所有的实时作业都运行在独立的集群上，不受外部的影响，这也方便后期对于Flink作业的开发与维护。后期通过Alluxio实现主集群与子集群间的数据共享。

在使用过程中，也遇到了形形色色不同的Bug，这边简单的介绍几个较为严重的问题。首先第一个问题是，Spark Streaming每个批次Job都会通过DirectKafkaInputStream的comput方法获取消费的Kafka Topic当前最新的offset，如果此时kafka集群由于某些原因不稳定，就会导致 java.lang.RuntimeException: No leader found for partition xx的问题，由于此段代码运行在Driver端，如果没有做任何配置和处理的情况下，会导致程序直接挂掉。对应的解决方法是配置spark.streaming.kafka.maxRetries大于1，并且可以通过配置refresh.leader.backoff.ms参数设置每次重试的间隔时间。

其次在使用Spark Streaming与Spark Sql相结合的过程中，也会有诸多问题。比如在使用过程中可能出现out of memory：PermGen space，这是由于Spark sql使用code generator

导致大量使用PermGen space，通过在spark.driver.extraJavaOptions中添加-XX:MaxPermSize=1024m -XX:PermSize=512m解决。还有Spark Sql需要创建Spark Warehouse，如果基于Yarn来运行，默认可能是在HDFS上创建相对应的目录，如果没有权限会报出Permission denied的问题，用户可以通过配置config("spark.sql.warehouse.dir", "file:${system:user.dir}/spark-warehouse")来解决。

未来展望

上面主要针对Spark Streaming在携程实时平台中的运用做了详细的介绍，在使用Spark Streaming过程中还是存在一些痛点，比如窗口功能比较单一、基于Event Time统计指标过于繁琐以及官方在新的版本中基本没有新的特性加入等，这使得我们更加倾向于尝试Flink。Flink基本实现了Google提出的各类实时处理的理念，引入了WaterMark的实现，感兴趣的朋友可以查看Google官方文档：The world beyond batch: Streaming 102。

目前Flink 1.4 release版本发布在即，Spark 2.2.0基于kafka数据源的Structured Streaming也支持了更多的特性。前期我们已对Flink做了充分的调研，下半年主要工作将放在Flink的对接上。在提供了诸多实时计算框架的支持后，随之而来的是带来了更多的学习成本，今后我们的重心将放在如何使用户更加容易的实现实时计算逻辑。其中Apache Beam对各种实时场景提供了良好的封装并对多种实时计算引擎做了支持，其次基于Stream Sql实现复杂的实时应用场景都将是我们主要调研的方向。