Apache Hudi 现在也支持 Flink 引擎了

本文作者：王祥虎，原文链接：https://mp.weixin.qq.com/s/LvKaj5ytk6imEU5Dc1Sr5Q
，欢迎关注 Apache Hudi 技术社区公众号：ApacheHudi。

Apache Hudi是由Uber开发并开源的数据湖框架，它于2019年1月进入Apache孵化器孵化，次年5月份顺利毕业晋升为Apache顶级项目。是当前最为热门的数据湖框架之一。

如果想及时了解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

为何要解耦

Hudi自诞生至今一直使用Spark作为其数据处理引擎。如果用户想使用Hudi作为其数据湖框架，就必须在其平台技术栈中引入Spark。放在几年前，使用Spark作为大数据处理引擎可以说是很平常甚至是理所当然的事。因为Spark既可以进行批处理也可以使用微批模拟流，流批一体，一套引擎解决流、批问题。然而，近年来，随着大数据技术的发展，同为大数据处理引擎的Flink逐渐进入人们的视野，并在计算引擎领域获占据了一定的市场，大数据处理引擎不再是一家独大。在大数据技术社区、论坛等领地，hudi是否支持使用flink计算引擎的的声音开始逐渐出现，并日渐频繁。所以使Hudi支持Flink引擎是个有价值的事情，而集成Flink引擎的前提是Hudi与Spark解耦。

同时，纵观大数据领域成熟、活跃、有生命力的框架，无一不是设计优雅，能与其他框架相互融合，彼此借力，各专所长。因此将Hudi与Spark解耦，将其变成一个引擎无关的数据湖框架，无疑是给Hudi与其他组件的融合创造了更多的可能，使得Hudi能更好的融入大数据生态圈。

解耦难点

Hudi内部使用Spark API像我们平时开发使用List一样稀松平常。自从数据源读取数据，到最终写出数据到表，无处不是使用Spark RDD作为主要数据结构，甚至连普通的工具类，都使用Spark API实现，可以说Hudi就是用Spark实现的一个通用数据湖框架，它与Spark的绑定可谓是深入骨髓。

此外，此次解耦后集成的首要引擎是Flink。而Flink与Spark在核心抽象上差异很大。Spark认为数据是有界的，其核心抽象是一个有限的数据集合。而Flink则认为数据的本质是流，其核心抽象DataStream中包含的是各种对数据的操作。同时，Hudi内部还存在多处同时操作多个RDD,以及将一个RDD的处理结果与另一个RDD联合处理的情况，这种抽象上的区别以及实现时对于中间结果的复用，使得Hudi在解耦抽象上难以使用统一的API同时操作RDD和DataStream。

解耦思路

理论上，Hudi使用Spark作为其计算引擎无非是为了使用Spark的分布式计算能力以及RDD丰富的算子能力。抛开分布式计算能力外，Hudi更多是把 RDD作为一个数据结构抽象，而RDD本质上又是一个有界数据集，因此，把RDD换成List,在理论上完全可行(当然，可能会牺牲些性能)。为了尽可能保证Hudi Spark版本的性能和稳定性。我们可以保留将有界数据集作为基本操作单位的设定，Hudi主要操作API不变，将RDD抽取为一个泛型，Spark引擎实现仍旧使用RDD，其他引擎则根据实际情况使用List或者其他有界数据集。

解耦原则：

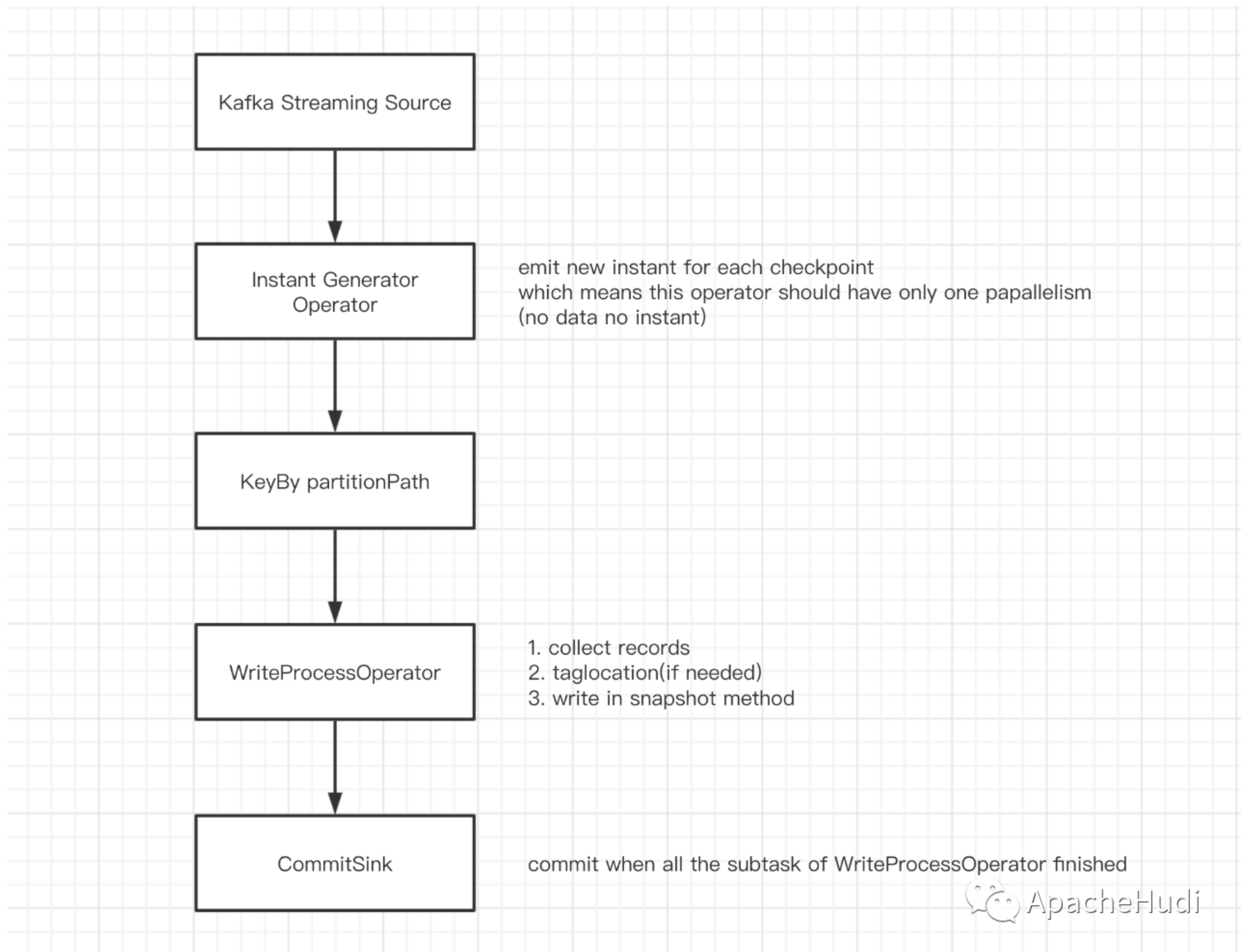
- 统一泛型。Spark API用到的JavaRDD,JavaRDD,JavaRDD统一使用泛型I,K,O代替；
- 去Spark化
。抽象层所有API必须与Spark无关。涉及到具体操作难以在抽象层实现的，改写为抽象方法，引入Spark子类实现。例如：Hudi内部多处使用到了JavaSparkContext#map()方法，去Spark化，则需要将JavaSparkContext隐藏，针对该问题我们引入了HoodieEngineContext#map()方法，该方法会屏蔽map的具体实现细节，从而在抽象成实现去Spark化。
- 抽象层尽量减少改动，保证Hudi原版功能和性能；
- 使用HoodieEngineContext抽象类替换JavaSparkContext，提供运行环境上下文。

Flink集成设计

Hudi的写操作在本质上是批处理，DeltaStreamer的连续模式是通过循环进行批处理实现的。为使用统一API，Hudi集成Flink时选择攒一批数据后再进行处理，最后统一进行提交(这里Flink我们使用List来攒批数据)。

攒批操作最容易想到的是通过使用时间窗口来实现，然而，使用窗口，在某个窗口没有数据流入时，将没有输出数据，Sink端难以判断同一批数据是否已经处理完。因此我们使用Flink的检查点机制来攒批，每两个Barrier之间的数据为一个批次，当某个子任务中没有数据时，mock结果数据凑数。这样在Sink端，当每个子任务都有结果数据下发时即可认为一批数据已经处理完成，可以执行commit。

DAG如下：



如果想及时了解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公众号：iteblog_hadoop

- source 接收kafka数据，转换成List;
- InstantGeneratorOperator 生成全局唯一的instant.当上一个instant未完成或者当前批次无数据时，不创建新的instant；
- KeyBy partitionPath 根据 partitionPath分区，避免多个子任务写同一个分区；
- WriteProcessOperator 执行写操作，当当前分区无数据时，向下游发送空的结果数据凑数；
- CommitSink 接收上游任务的计算结果，当收到 parallelism个结果时，认为上游子任务全部执行完成，执行commit.

注：InstantGeneratorOperator和WriteProcessOperator 均为自定义的Flink算子，InstantGeneratorOperator会在其内部阻塞检查上一个instant的状态，保证全局只有一个inflight（或requested）状态的instant.WriteProcessOperator是实际执行写操作的地方，其写操作在checkpoint时触发。

实现示例

HoodieTable

```
/**
 * Abstract implementation of a HoodieTable.
 *
 * @param <T> Sub type of HoodieRecordPayload
 * @param <I> Type of inputs
 * @param <K> Type of keys
 * @param <O> Type of outputs
 */
public abstract class HoodieTable<T extends HoodieRecordPayload, I, K, O> implements Serializable {

  protected final HoodieWriteConfig config;
  protected final HoodieTableMetaClient metaClient;
  protected final HoodieIndex<T, I, K, O> index;

  public abstract HoodieWriteMetadata<O> upsert(HoodieEngineContext context, String instantTime,
      I records);

  public abstract HoodieWriteMetadata<O> insert(HoodieEngineContext context, String instantTime,
      I records);

  public abstract HoodieWriteMetadata<O> bulkInsert(HoodieEngineContext context, String instantTime,
      I records, Option<BulkInsertPartitioner<I>> bulkInsertPartitioner);

  ......
}
```

HoodieTable 是 hudi的核心抽象之一，其中定义了表支持的insert,upsert,bulkInsert等操作。以 upsert 为例，输入数据由原先的 JavaRDD<hoodierecord> inputRdds 换成了 I records, 运行时 JavaSparkContext jsc 换成了 HoodieEngineContext context.

从类注释可以看到 T,I,K,O分别代表了hudi操作的负载数据类型、输入数据类型、主键类型以及输出数据类型。这些泛型将贯穿整个抽象层。

HoodieEngineContext

```
/**
```

```
 * Base class contains the context information needed by the engine at runtime. It will be extended by different
 * engine implementation if needed.
 */
public abstract class HoodieEngineContext {

  public abstract <I, O> List<O> map(List<I> data, SerializableFunction<I, O> func, int parallelism);

  public abstract <I, O> List<O> flatMap(List<I> data, SerializableFunction<I, Stream<O>> func, int parallelism);

  public abstract <I> void foreach(List<I> data, SerializableConsumer<I> consumer, int parallelism);

  ......
}
```

HoodieEngineContext 扮演了 JavaSparkContext 的角色，它不仅能提供所有 JavaSparkContext能提供的信息，还封装了 map,flatMap,foreach等诸多方法，隐藏了JavaSparkContext#map()，JavaSparkContext#flatMap()，JavaSparkContext#foreach()等方法的具体实现。

以map方法为例，在Spark的实现类 HoodieSparkEngineContext中，map方法如下：

```
  @Override
  public <I, O> List<O> map(List<I> data, SerializableFunction<I, O> func, int parallelism) {
    return javaSparkContext.parallelize(data, parallelism).map(func::apply).collect();
  }
```

在操作List的引擎中其实现可以为（不同方法需注意线程安全问题，慎用parallel()）：

```
  @Override
  public <I, O> List<O> map(List<I> data, SerializableFunction<I, O> func, int parallelism) {
    return data.stream().parallel().map(func::apply).collect(Collectors.toList());
  }
```

注：map函数中抛出的异常，可以通过包装SerializableFunction<i , O> func解决.

这里简要介绍下 SerializableFunction:

```
@FunctionalInterface
public interface SerializableFunction<I, O> extends Serializable {
  O apply(I v1) throws Exception;
}
```

该方法实际上是 java.util.function.Function 的变种，与java.util.function.Function 不同的是 SerializableFunction可以序列化，可以抛异常。引入该函数是因为JavaSparkContext#map()函数能接收的入参必须可序列，同时在hudi的逻辑中，有多处需要抛异常，而在Lambda表达式中进行 try catch 代码会略显臃肿，不太优雅。

现状和后续计划

工作时间轴

- 2020年4月，T3出行（杨华@vinoyang，王祥虎@wangxianghu）和阿里巴巴的同学（李少锋@leesf）以及若干其他小伙伴一起设计、敲定了该解耦方案；
- 2020年4月，T3出行(王祥虎@wangxianghu)在内部完成了编码实现，并进行了初步验证，得出方案可行的结论；
- 2020年7月，T3出行(王祥虎@wangxianghu)将该设计实现和基于新抽象实现的Spark版本推向社区（HUDI-1089）；
- 2020年9月26日，顺丰科技基于T3内部分支修改完善的版本在 Apache Flink Meetup（深圳站）公开PR, 使其成为业界第一个在线上使用Flink将数据写Hudi的企业。
- 2020年10月2日，HUDI-1089 合并入Hudi主分支，标志着Hudi-Spark解耦完成。

后续计划

1）推进Hudi和Flink集成

将Flink与Hudi的集成尽快推向社区，初期该特性可能只支持kafka数据源。

2）性能优化

为保证Hudi-Spark版本的稳定性和性能，此次解耦没有太多考虑Flink版本可能存在的性能问题。

3）类flink-connector-hudi第三方包开发

将Hudi-Flink的绑定做成第三方包，用户可以在Flink应用中以编码方式读取任意数据源，通过这个第三方包写入Hudi。

本博客文章除特别声明，全部都是原创！

原创文章版权归过往记忆大数据（过往记忆）所有，未经许可不得转载。
本文链接: 【】（）