重启Job时遇到NoClassDefFoundError

问题描述

在刚开始运行Rheos SQL的程序时，同一个class重启的时候，会出现NoClassDfnFoundError[1]，这个Error的cause是ClassNotFoundException。真正的stack trace没有保留下来，类似的如下：

```
java.lang.NoClassDefFoundError: com/ebay/integ/dal/cm/smartcp/SmartConnectionPool$ErlangCalculator
        at com.ebay.integ.dal.cm.smartcp.SmartConnectionPool.evaluatePoolSize(SmartConnectionPool.java:1378)
        at com.ebay.integ.dal.cm.smartcp.SmartConnectionPool.access$2300(SmartConnectionPool.java:69)
        at com.ebay.integ.dal.cm.smartcp.SmartConnectionPool$3.run(SmartConnectionPool.java:404)
        at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:511)
        at java.util.concurrent.FutureTask.runAndReset(FutureTask.java:308)
        at
java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask.access$301(ScheduledThreadPoolExecutor.java:180)
        at
java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask.run(ScheduledThreadPoolExecutor.java:294)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)
        at java.lang.Thread.run(Thread.java:748)
Caused by: java.lang.ClassNotFoundException: com.ebay.integ.dal.cm.smartcp.SmartConnectionPool$ErlangCalculator
        at java.net.URLClassLoader.findClass(URLClassLoader.java:381)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:424)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:357)
        ... 10 more
```

将package下载到本地解压后，发现class是在的。

Debug过程

从stackTrace来看，错误是从URLClassLoader的findClass方法抛出来的，进到源码里，可以看到下面这段注释：

** @exception ClassNotFoundException if the class could not be found,*

** or **if the loader is closed**.*

其中加粗的部分，是ClassLoader的方法签名中不从提到的情况。在ClassLoader类中，也没有close方法。然而，URLClassLoader的类签名中，实现了Closeable接口。在URLClassLoader的close方法注释中，有下面这段：

** Closes this URLClassLoader, so that it can no longer be used to load*

** new classes or resources that are defined by this loader.*

所以，问题可能是类被加载过，但是加载这个类的classLoader被调用了close方法。

通常，JVM中加载的类，不会随意回收，ClassLoader也不应该被关闭，因为JVM很难知道一个已经被加载的类在什么情况下可以定义成完全不需要。难道是Flink本身有什么特别之处？简单搜索后，可以发现Flink有自己定制过的类加载器[2]。

从JVM的角度看，类加载器分为下面三类：

- Bootstrap Classloader：JVM自带，加载${java_home}/lib下的package，C++实现。
- Extension Classloader：加载${jave_home}/lib/ext目录下的package，Java实现。
- Application Classloader：getSystemClassloader()的返回值，加载用户Classpath上指定的类，程序中默认使用的类加载器，java实现。

作为一个Java application，Flink进程本身是long running的，需要的类由Application Classloader加载，主要是Flink lib/目录下的包。
对于Flink plugins/目录和用户动态提交上来的job，flink是动态的加载和卸载的。使用的classloader是FlinkUserCodeClassLoaders。这个ClassLoader的parent，是load这个类本身的ClassLoader，也就是Application Classloader。
根据Flink Cluster配置的不同，FlinkUserCodeClassLoaders有两种加载方式，parent-first和child-first，两种实现都是URLClassLoader的子类。为了使用户代码有更好的隔离性，默认使用的是child-first的方式，也就是需要加载一个类时，优先从用户的包里找，找不到再往上代理。对于那些Flink framework本身和用户代码都会应用到的类，比如“org.apache.flink”下的，会作为特殊情况处理，依然保持parent-first。用户可以通过配置，来关掉是FlinkUserCodeClassLoaders child-first的行为，改用传统的parent-first。

回到最初的问题，抛错的Class是在Job的package里，会被FlinkUserCodeClassLoaders，在首次运行时，可以被正确加载，当重启时，当前Job的Classloader会被close，那么再去引用由当前job的classloader加载的类时，就会抛出上述看到的错误。这种对类的引用，发生在一些没有被成功关闭的资源中，比如没有被shutdown的thread。

解决方案

这个问题最终是一个依赖的实现有问题，当升级他的版本之后，问题修复了。

扩展

这篇文章[3]讲述的问题，跟我们遇到的十分类似。其中提到了一个叫bTrace[4]的tool，可以使用script将类和加载他的class一起打印出来，示例脚本如下：

```
package test;

import com.sun.btrace.annotations.*;
import static com.sun.btrace.BTraceUtils.*;

@BTrace
public class FindClassOnClosedLoader {
  @OnMethod(clazz="+java.net.URLClassLoader", method="findClass")
  public static void onFindClass(
     @Self Object o,
     @ProbeClassName String probeClass,
     @ProbeMethodName String probeMethod,
     Object arg) {
    println("Executing " + probeClass + "." + probeMethod + "(" + arg +
                ") on " + o);
  }
```

```
  @OnMethod(clazz="+java.net.URLClassLoader", method="close")
  public static void onClose(
      @Self Object o,
      @ProbeClassName String probeClass,
      @ProbeMethodName String probeMethod) {
    println("Executing " + probeClass + ".close() on " + o);
  }
}
```

示例输出：

Executing java.net.URLClassLoader.findClass(org.apache.flink.runtime.execution.librarycache.FlinkUserCodeClassLoaders$ChildFirstClassLoader) on sun.misc.Launcher$ExtClassLoader@6956de9

Executing java.net.URLClassLoader.findClass(org.apache.flink.runtime.execution.librarycache.FlinkUserCodeClassLoaders$ChildFirstClassLoader) on sun.misc.Launcher$AppClassLoader@7d4991ad

Support GenericRecord出错

问题描述

在内部的一个use case中，需要将source里的record一部分，直接透传到sink，中间不做任何改动。在schema中，这部分的类型是GenericRecord，所以想在rheos sql的类型系统中支持GenericRecord。但是，在集群上运行时，抛出了以下错误：

```
com.esotericsoftware.kryo.KryoException: Error constructing instance of class: org.apache.avro.Schema$LockableArrayList
Serialization trace:
types (org.apache.avro.Schema$UnionSchema)
schema (org.apache.avro.Schema$Field)
fieldMap (org.apache.avro.Schema$RecordSchema)
schema (org.apache.avro.generic.GenericData$Record)
	at
。。。
com.twitter.chill.Instantiators$$anon$1.newInstance(KryoBase.scala:136)
	at com.esotericsoftware.kryo.Kryo.newInstance(Kryo.java:1061)
	at Caused by: java.lang.IllegalAccessException: Class com.twitter.chill.Instantiators$$anonfun$normalJava$1 can not access a member of class org.apache.avro.Schema$LockableArrayList with modifiers "public"
	at sun.reflect.Reflection.ensureMemberAccess(Reflection.java:102)
	at java.lang.reflect.AccessibleObject.slowCheckMemberAccess(AccessibleObject.java:296)
	at java.lang.reflect.AccessibleObject.checkAccess(AccessibleObject.java:288)
	at java.lang.reflect.Constructor.newInstance(Constructor.java:413)
```

```
	at
com.twitter.chill.Instantiators$$anonfun$normalJava$1.apply(KryoBase.scala:170)
	at
com.twitter.chill.Instantiators$$anon$1.newInstance(KryoBase.scala:133)
	... 40 more
```

Debug过程

从Error trace上看，最易想到的原因是，avro版本冲突，“LockableArrayList”的可见性在不同版本有差异。但是仔细看了dependency tree，并没有这种情况。
抛错中提到了另外一种serialization的方式Kryo，看上去问题与Flink的类型系统和serialization有关，官方文档[5]里也确实解释了这方面的内容。
首先，作为一套流处理框架，Flink需要在operator之间不断地传递数据，所以对于类型的感知和网络传输，是FLink关注的问题。对于基本类型，FLink原生可以支持，对于Pojo类型和其他的复杂类型，用户有几个不同的选择：

- Flink默认的行为，会交给Kryo[5]
- 用户为特定的类型自定义一个serializer，并注册到ExecutionEnvironment里
- 明确的设置Flink使用Avro系统作为Serializer

对于要支持的GenericRecord，我们选取第三种解决方案是最方便的。

解决方案

要强制使用Avro系统的serializer，需要添加flink-avro的依赖，然后加入这段代码：

```
final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();
env.getConfig().enableForceAvro();
```

对于一些flink的版本，还需要将classloader显示指定为parent-first。这可能是flink的bug造成的，有一些讨论可以参照[6].

eBay-items的sql版本performance很差

问题描述

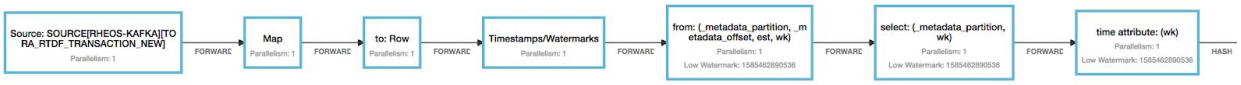
在将storm版本的ebay items移植成rheos sql之后，发现performance变得很差。单个slot，正常来讲，每秒可以处理4K左右的数据，但是现在只能处理几十个。
Job的逻辑是，读kafka的source，跟oracle的表进行一次side join，然后输出到Kafka的sink。在Chain过之后，只生成两个task。
如果添加大量的节点和并行度，最终还是可以支持当前的数据量，但是最好还是要知道为什么性能差别这么大。

Debug过程

由于job的逻辑中，使用了外界的Oracle做join，所以优先怀疑Oracle是整套系统的瓶颈。主要做两方面的工作来验证这一点：

- 增加异步线程池大小和连接池大小，但是性能并没有明显的提升
- 添加更加细粒度的监控，暴露链接获取，sql执行和链接释放的开销，发现在可接受范围内

看来是Job pipeline中的其他步骤出现问题。为了找到瓶颈，关掉了Flink pipeline里的chain，每一个operator变成单独的task。
Job的execution graph会被拉成类似下图的结构：



观察Flink的backpressure页面，发现从source之后的Map task开始，都有非常严重的背压。关于Flink背压的原理，可以参照这篇[7]博客。
Flink的task之间传递数据时，需要从LocalBufferPool中申请Buffer，使用完了再返回回去。如果一个task里对数据的处理很慢，将Buffer还回到LocalBufferPool就会很慢，线程会卡在从LocalBufferPool中申请Buffer的阶段，stack trace如下：

```
java.lang.Object.wait(Native Method)
o.a.f.[...].LocalBufferPool.requestBuffer(LocalBufferPool.java:163)
o.a.f.[...].LocalBufferPool.requestBufferBlocking(LocalBufferPool.java:133)
<--- BLOCKING request
[...]
```

Flink就是通过检查Flink task的stack trace是不是满足这个条件，来判定是不是发生了背压。
本地Debug，可以看到Map的逻辑很简单，是将Source的输出Row转为CRow，CRow是Row的包装类，多添加了boolean类型的成员变量，这个过程应该很快速才对。
修改有几个思路，其中最容易想到的一个，是将Source的输出改为cRow，这样Map operator就不需要了。这种做法的问题是，cRow并不是Flink SQL原生支持的类型，所以需要额外的逻辑处理，整套系统里，使用Row作为最基本的传递类型，要修改的地方特别多，另外，Map的逻辑已经足够简单，省略掉这个operator，可能只是进一步的隐藏真正的问题。
接下来想到的是，是不是LocalBufferPool不够大，这个文档[8]中记录了如何调节参数。在将buffer的大小变大很多后，依然没有效果，buffer被用完后，topology又开始背压了。
那只剩下一种可能，网络中传输的数据量太大。在我们的实现中，会将source中的所有域展开，在几次删减测试之后，发现只要将一个类型为GenericRecord的域加进去，performance就开始下降。GenericRecord本身，关联的对象很多，其中最大的，是Schema对象。应该是在GenericRecord被序列化的时候，变得很大。
为了验证这一发现，在Metrics中添加输出message的平均大小:
flink_taskmanager_job_task_numBytesOutPerSecond / flink_taskmanager_job_task_numRecordsOutPerSecond

当输出中不包含GenericRecord时，平均大小在1K一下，但是当包含GenericRecord，大小会上升到40K以上。

解决方案

解决方案十分简单。由于在整条数据流中，并不会解析GenericRecord，所以我们最终决定在source的输出中，将GenericRecord转化成byte[]传递给下游，在sink输出的时候，再将byte[]转化成GenericRecord。

Avro serializer造成的Count distinct失效

问题描述

```
INSERT INTO
  Console
SELECT
  TUMBLE_START(wk, INTERVAL '10' second) as window_start,
  count(distinct _metadata_partition) as ct
  FROM TORA_RTDF_TRANSACTION_NEW
GROUP BY TUMBLE(wk, INTERVAL '10' second);
```

在支持了Avro format并且强制指定Pojo的secrializer是Avro之后，发现count(distinct $field)会抛错，ErrorTrace如下：

```
org.apache.avro.AvroTypeException: Unknown type: ACC
    at org.apache.avro.specific.SpecificData.createSchema(SpecificData.java:255)
    at org.apache.avro.reflect.ReflectData.createSchema(ReflectData.java:514)
    at org.apache.avro.reflect.ReflectData.createFieldSchema(ReflectData.java:593)
    at org.apache.avro.reflect.ReflectData.createSchema(ReflectData.java:472)
    at org.apache.avro.specific.SpecificData.getSchema(SpecificData.java:189)
    at org.apache.flink.formats.avro.typeutils.AvroFactory.fromReflective(AvroFactory.java:123)
    at org.apache.flink.formats.avro.typeutils.AvroFactory.create(AvroFactory.java:86)
    at org.apache.flink.formats.avro.typeutils.AvroSerializer.initializeAvro(AvroSerializer.java:330)
    at org.apache.flink.formats.avro.typeutils.AvroSerializer.checkAvroInitialized(AvroSerializer.java:325)
    at org.apache.flink.formats.avro.typeutils.AvroSerializer.duplicate(AvroSerializer.java:286)
    at org.apache.flink.api.java.typeutils.runtime.RowSerializer.duplicate(RowSerializer.java:74)
    at org.apache.flink.api.common.state.StateDescriptor.getSerializer(StateDescriptor.java:195)
    at org.apache.flink.runtime.state.heap.HeapKeyedStateBackend.tryRegisterStateTable(HeapKeyedStateBackend.java:260)
    at org.apache.flink.runtime.state.heap.HeapKeyedStateBackend.createInternalState(HeapKeyedStateBackend.java:341)
    at org.apache.flink.runtime.state.KeyedStateFactory.createInternalState(KeyedStateFactory.java:47)
    at org.apache.flink.runtime.state.ttl.TtlStateFactory.createStateAndWrapWithTtlIfEnabled(TtlStateFactory.java:63)
    at org.apache.flink.runtime.state.AbstractKeyedStateBackend.getOrCreateKeyedState(AbstractKeyedStateBackend.java:241)
    at org.apache.flink.streaming.api.operators.AbstractStreamOperator.getOrCreateKeyedState(AbstractStreamOperator.java:568)
    at org.apache.flink.streaming.runtime.operators.windowing.WindowOperator.open(WindowOperator.java:240)
    at org.apache.flink.streaming.runtime.tasks.StreamTask.openAllOperators(StreamTask.java:424)
    at org.apache.flink.streaming.runtime.tasks.StreamTask.invoke(StreamTask.java:290)
    at org.apache.flink.runtime.taskmanager.Task.run(Task.java:704)
    at java.lang.Thread.run(Thread.java:748)
```

Debug过程

从StackTrace中，比较明确的可以知道问题与Avro serializer有关系，但在单独debug这个问题之前，我想先研究一下window的aggregation是怎么做的。

这段SQL语句翻译成运行时的operator，是由WindowOperator来执行的。初始化的时候，会构造出一个windowState的对象，后续processElement方法大概逻辑是：

- 根据element的timestamp，确定出一个当前应该操作的窗口
- 将当前的element value，add到windowState
- 判定是否满足trigger的条件，如果满足，从windowState将聚合结果取出来，emit出去

所以，主体的aggregation逻辑，是在windowState中做的。windowState会关联一个StateDescriptor，从外到内，被引用到的接口和在我们例子中的实现类如下：

- StateDescriptor：AggregatingStateDescriptor
- AggregateFunction: AggregateAggFunction
- GeneratedAggregationsFunction: 根据SQL codeGen出的function

主体的流程是，在State中调用AggregateFunction的createAccumulators()方法，获取一个用于实现计算逻辑的ACC，然后在State的add方法中，调用到AggregateFunction的

ACC add(IN value, ACC accumulator)

方法，将新到达的value利用ACC计算。
AggregateFucntion的createAccumulators方法，会最终调用到Gen出来的Func的createAccumulators方法，在当前例子中，该方法返回了只有一列的Row，列的类型是DistinctAccumulator。是DistinctAccumulator这个类内部维护了一个MapView，用来实现根据输入distinct的逻辑，这个类初始化的时候会传入CountAccumulator，来真正的做aggregation的计算。
AggregateFunction的Add方法，会最终调用到Gen出来的Func的accumulate方法。当前例子中，该方法的逻辑如下：

- 从ACC中果DistinctAccumulator取出来
- 如果DistinctAccumulator的add方法在input上返回true，也就是这确实是一个新的input，就继续，否则直接返回了
- 从DistinctAccumulator中，将CountAccumulator取出来
- 在CountAccumulator上加1

以上是在window中做Aggregation的过程，当window满足fire条件，在windowState上调用get方法时，会最终去到CountAccumulator中将结果取出。

回到最开始遇到的问题。由于Avro的serialization需要Schema，而Row中使用的DistinctAccumulator并不继承自GenericRecord或者其他Avro常见的对象，最终会使用反射构造Schema。DistinctAccumulator有一个type是”ACC”的成员变量叫realAcc，ACC是个模板类型，在我们例子中，“ACC”会在运行时被实例化成CountAccumulator，但是在生成Schema的时候，”ACC"被Field.getGenericType()作为一种类型返回回去，也就有了上面的error message。

由此可见，如果使用Avro作为默认的POJO的serializer，当成员变量中出现模板类型，都有可能会触发上面的运行时错误。

解决方案

在遇到上个问题时，我们已经放弃使用GenericRecord，所以也不需要继续支持Avro，最终把当初支持Avro的修改rollback，问题得到了fix。

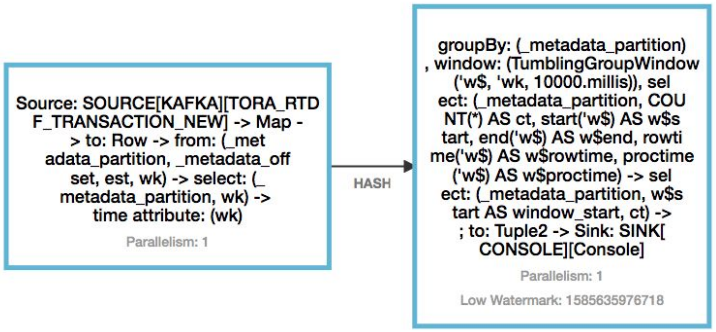
Rheos SQL job从savepoint启动watermark不回退

问题描述

用户从一个已有系统移植到Rheos SQL之后，尝试从过去的savepoint启动，与原有系统比较时，发现有数据丢失。Job的主体逻辑是，读Kafka的source，在event timestamp上group by，然后输出结果到某个sink，在调试的时候，sink是stdout。
主体SQL脚本如下：

```
INSERT
INTO
                Console
            SELECT
                _metadata_partition,
                TUMBLE_START(wk, INTERVAL '10' second) as window_start,
                count(*) as ct
                FROM TORA_RTDF_TRANSACTION_NEW
            GROUP BY _metadata_partition, TUMBLE(wk, INTERVAL '10' second);
```

job的execution graph如下：

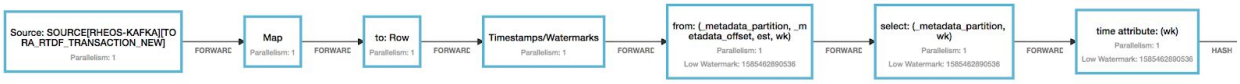


从metrics的表现上看，第二个task的watermark，不会因为从savepoint的启动而回退到过去的时间，而且，有很多lateness records被丢弃掉了。

Debug过程

首先，需要弄清楚数据是怎样被判定成lateness records的，metrics的全名是：

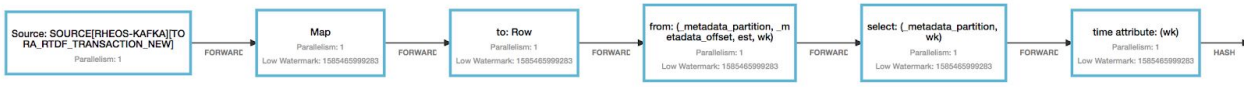
flink_taskmanager_job_task_operator_numLateRecordsDropped
这个metrics是从WindowOperator中暴露出来的。判断逻辑在isElementLate方法中，如果当前window使用的是event timestamp并且event.timestamp加allowLateness比当前watermark小，就会被mark成lateness record。
理论上说，Kafka consumer的状态回退到之前，而watermark没有，才会造成当前的情况。为了验证kafka consumer确实回退了，在阅读源码之后，发现FlinkKafkaConsumerBase.initializeState方法中，会打印出一条log：
Consumer subtask {} restored state: {}.
在我们flink cluster的log中，确实发现了这段log，而且重复start from Savepoint，起始的offset都是一样的，所以可以断定，在重启的过程中，kafka的offset确实回退并重放数据了。
Watermark没有回退，可能有两个原因，第一个是watermark作为state的一部分，被打到savepoint中了，重启的时候，watermark是从state中读的。细想一下，这并不太合理，因为savepoint是依次在operator中流转的，watermark即使被保存下来，应该也是跟kafka source的state相一致。在查看源码后，发现watermark并不会作为state的一部分被保存，社区甚至有一个ticket[9]是讨论怎么样可以做到这一点的。第二种可能是watermark的计算有问题。当一个operator的input有多个时，emit出去的watermark一定是所有input中可见的最小值。对于Kafka来说，input是各个partition。根据partition来emit watermark的逻辑，在AbstractFetcher的PeriodicWatermarkEmitter。只有当watermark的模式是PERIODIC_WATERMARKS时，PeriodicWatermarkEmitter才会被调用，WatermarkMode是根据FlinkKafkaConsumerBase中的watermarkAssigner决定的。
这个时候，我们意识到在rheos-sql中，watermarkAssigner并不是注入到FlinkKafkaConsumer中，而是接在Source返回的DataStream上，将Chain disable掉之后，可以看到下面这个executionGraph：



可以明确的看出，在source之后的Timestamp/Watermarks task中，才会产生watermark，并且后续的task才会获得watermark。而在当前watermark产生中，并没有根据partition等待的逻辑，而是只是产生一个在某一小段时间内的最小的event timestamp而已。对于这个task，只有一个输入，就是上游的to:Row task。来自不同partition的数据，在经历了Map和to:Row之后，都被混杂在一起，输入到Timestamp/Watermarks task。
当Job从savepoint启动，kafka source中有lag，很有可能出现某些partition读取数据比较快的情况，这些来自某个partition的数据，会讲watermark的值提高很多，那么后面来自于其他partition的数据，就很有可能被当做lateness record了。这些数据就丢失了。
Dashboard中的watermark，其实并不是没有回退，而是因增长太快，当显示到dashboard上时，已经变得很大了。
再回头观察一下输出，当重启job的时候，每次读的比较快的partition都可能不一样，但是watermark确实跟着这些读的快的partition一起增长了。

解决方案

要fix这个问题，最简单的办法是把watermark assigner从source之外移到source当中。生成出来的新的execution group是：



可以看到，之前的watermark/Timestamps task不见了，而且从source之后的所有task都拥有了watermark。

这个解决方案并不完美，因为当source的各个partition中数据非常不均匀，甚至有些partition没有数据时，会造成watermark一直不前进。大概的解决思路是，设置某种超时机制，当一些partition自己的watermark长时间不往前近的时候，就忽略掉。

社区中，很久之前就开始有ticket[10]讨论这个问题，但是迟迟没有解决。最终，这个问题会在新的source design[11]中被处理。

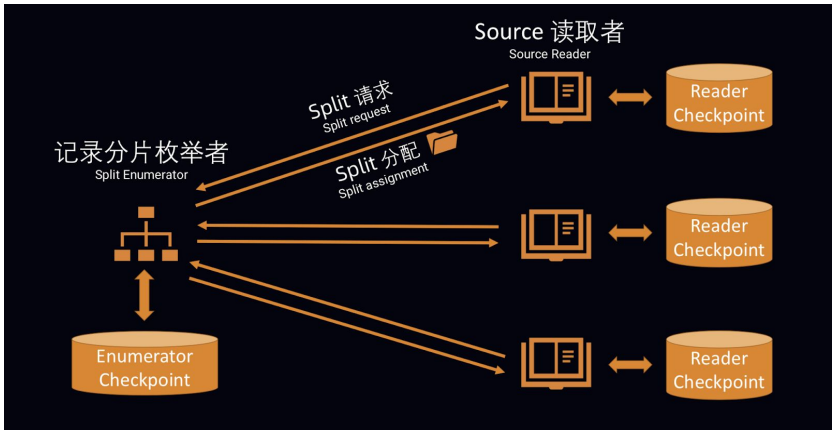
现在source的问题有：

- Batch和stream模式不统一，而且在Stream模式下，并没有中心化的模块来协同source分片的分配和处理。
- 不同的source之间，实现没法公用，需要单独做一套。比如Kafka对于watermark的处理，并不能直接被Pulsar使用。
- 线程模型复杂，而且与新版本的一些模型不兼容。

在新的source设计中，定义了三种核心的抽象：

- Source split：包含了数据分片的所有信息，读取进度可以被写入到checkpoint。
- Source Reader：用来读取source split，并且产生watermark
- Split Enumrator：集中式模块，发现source split，并且协调各个source reader之间的任务分配

结构如下：



每个split的watermark是独立生成的，有三种情况：

- 当有新的record到来时生成
- 周期性的生成
- 当这个split空闲时生成，我们上面讨论的情况会被这个方式cover

对于source整体的watermark，交给统一的enumerator来完成。
当前，对于新功能接口的定义已经merge到了master，但是完整的功能需要等待社区后面的release。

Reference

[1] https://dzone.com/articles/java-classnotfoundexception-vs-noclassdeffounderro
[2] https://ci.apache.org/projects/flink/flink-docs-stable/monitoring/debugging_classloading.html
[3] https://heap.io/blog/engineering/missing-scala-class-noclassdeffounderror
[4] https://github.com/btraceio/btrace/blob/master/docs/BTraceTutorial.md
[5] https://github.com/EsotericSoftware/kryo
[6] http://apache-flink-user-mailing-list-archive.2336050.n4.nabble.com/AvroInputFormat-Serialisation-Issue-td20146.html
[7] http://wuchong.me/blog/2016/04/26/flink-internals-how-to-handle-backpressure/
[8] https://ci.apache.org/projects/flink/flink-docs-release-1.10/ops/config.html#taskmanager-memory-network-fraction
[9] https://github.com/apache/flink/pull/7013
[10] https://issues.apache.org/jira/browse/FLINK-5479
[11] https://issues.apache.org/jira/browse/FLINK-10740