Flink DataStream API 中的多面手——Process Function

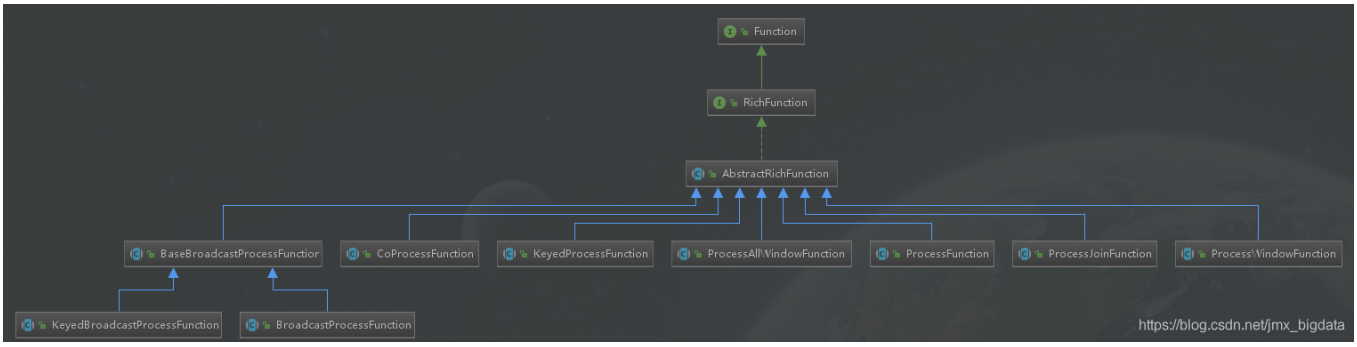
在Flink的时间与watermarks详解这篇文章中，阐述了Flink的时间与水位线的相关内容。你可能不禁要发问，该如何访问时间戳和水位线呢？首先通过普通的DataStream API是无法访问的，需要借助Flink提供的一个底层的API——Process Function。Process Function不仅能够访问时间戳与水位线，而且还可以注册在将来的某个特定时间触发的计时器(timers)。除此之外，还可以将数据通过Side Outputs发送到多个输出流中。这样以来，可以实现数据分流的功能，同时也是处理迟到数据的一种方式。下面我们将从源码入手，结合具体的使用案例来说明该如何使用Process Function。

简介

Flink提供了很多Process Function，每种Process Function都有各自的功能，这些Process Function主要包括：

- ProcessFunction
- KeyedProcessFunction
- CoProcessFunction
- ProcessJoinFunction
- ProcessWindowFunction
- ProcessAllWindowFunction
- BaseBroadcastProcessFunction
  - KeyedBroadcastProcessFunction
- BroadcastProcessFunction

继承关系图如下：



从上面的继承关系中可以看出，都实现了RichFunction接口，所以支持使用`open()`、`close()`、`getRuntimeContext()`等方法的调用。从名字上可以看出，这些函数都有不同的适用场景，但是基本的功能是类似的，下面会以KeyedProcessFunction为例来讨论这些函数的通用功能。

源码

KeyedProcessFunction

```
/**
 * 处理KeyedStream流的低级API函数
 * 对于输入流中的每个元素都会触发调用processElement方法.该方法会产生0个或多个输出.
 * 其实现类可以通过Context访问数据的时间戳和计时器(timers).当计时器(timers)触发时
, 会回调onTimer方法.
 * onTimer方法会产生0个或者多个输出, 并且会注册一个未来的计时器.
 *
 * 注意: 如果要访问keyed state和计时器(timers), 必须在KeyedStream上使用KeyedPro
cessFunction.
 * 另外, KeyedProcessFunction的父类AbstractRichFunction实现了RichFunction接
口, 所以, 可以使用
 * open(), close()及getRuntimeContext()方法.
 *
 * @param <K> key的类型
 * @param <I> 输入元素的数据类型
 * @param <O> 输出元素的数据类型
 */
@PublicEvolving
public abstract class KeyedProcessFunction<K, I, O> extends AbstractRich
Function {

        private static final long serialVersionUID = 1L;
        /**
         * 处理输入流中的每个元素
         * 该方法会输出0个或者多个输出, 类似于FlatMap的功能
         * 除此之外, 该方法还可以更新内部状态或者设置计时器(timer)
         * @param value 输入元素
         * @param ctx  Context, 可以访问输入元素的时间戳, 并其可以获取一个时间服
务器(TimerService), 用于注册计时器(timers)并查询时间
         *  Context只有在processElement被调用期间有效.
         * @param out  返回的结果值
         * @throws Exception
         */
        public abstract void processElement(I value, Context ctx, Collec
tor<O> out) throws Exception;

        /**
         * 是一个回调函数, 当在TimerService中注册的计时器(timers)被触发时, 会回调

```

```
该函数
         * @param timestamp 触发计时器(timers)的时间戳
         * @param ctx  OnTimerContext，允许访问时间戳，TimeDomain枚举类提供了
两种时间类型：
         * EVENT_TIME与PROCESSING_TIME
         * 并其可以获取一个时间服务器(TimerService)，用于注册计时器(timers)并查
询时间
         * OnTimerContext只有在onTimer方法被调用期间有效
         * @param out 结果输出
         * @throws Exception
         */
        public void onTimer(long timestamp, OnTimerContext ctx, Collecto
r<O> out) throws Exception {}
        /**
         * 仅仅在processElement()方法或者onTimer方法被调用期间有效
         */
        public abstract class Context {

                /**
                 * 当前被处理元素的时间戳，或者是触发计时器(timers)时的时间戳
                 * 该值可能为null，比如当程序中设置的时间语义为：TimeCharacteri
stic#ProcessingTime
                 * @return
                 */
                public abstract Long timestamp();

                /**
                 * 访问时间和注册的计时器(timers)
                 * @return
                 */
                public abstract TimerService timerService();

                /**
                 * 将元素输出到side output (侧输出)
                 * @param outputTag 侧输出的标记
                 * @param value 输出的记录
                 * @param <X>
                 */
                public abstract <X> void output(OutputTag<X> outputTag,
X value);
                /**
                 * 获取被处理元素的key
                 * @return
                 */
                public abstract K getCurrentKey();
        }
        /**
         * 当onTimer方法被调用时，才可以使用OnTimerContext
         */
```

```
89          public abstract class OnTimerContext extends Context {
                /**
                 * 触发计时器(timers)的时间类型，包括两种：EVENT_TIME与PROCESS
    ING_TIME
                 * @return
                 */
                public abstract TimeDomain timeDomain();
                /**
                 * 获取触发计时器(timer)元素的key
                 * @return
                 */
                @Override
                public abstract K getCurrentKey();
            }
    }
```

上面的源码中，主要有两个方法，分析如下：

- processElement(I value, Context ctx, Collector out)

该方法会对流中的每条记录都调用一次，输出0个或者多个元素，类似于FlatMap的功能，通过Collector将结果发出。除此之外，该函数有一个Context 参数，用户可以通过Context 访问时间戳、当前记录的key值以及TimerService(关于TimerService，下面会详细解释)。另外还可以使用output方法将数据发送到side output，实现分流或者处理迟到数据的功能。

- onTimer(long timestamp, OnTimerContext ctx, Collector out)

该方法是一个回调函数，当在TimerService中注册的计时器(timers)被触发时，会回调该函数。其中`@param timestamp`参数表示触发计时器(timers)的时间戳，Collector可以将记录发出。细心的你可能会发现，这两个方法都有一个上下文参数，上面的方法传递的是Context 参数，onTimer方法传递的是OnTimerContext参数，这两个参数对象可以实现相似的功能。OnTimerContext还可以返回触发计时器的时间域(EVENT_TIME与PROCESSING_TIME)。

TimerService

在KeyedProcessFunction源码中，使用TimerService来访问时间和计时器，下面来看一下源码：

```
1  @PublicEvolving
2  public interface TimerService {
3          String UNSUPPORTED_REGISTER_TIMER_MSG = "Setting timers is only
4  supported on a keyed streams.";
5          String UNSUPPORTED_DELETE_TIMER_MSG = "Deleting timers is only s
```

```
upported on a keyed streams.";
        // 返回当前的处理时间
        long currentProcessingTime();
        // 返回当前event-time水位线(watermark)
        long currentWatermark();

        /**
         * 注册一个计时器(timers), 当processing time的时间等于该计时器时钟时会被
调用
         * @param time
         */
        void registerProcessingTimeTimer(long time);

        /**
         * 注册一个计时器(timers),当event time的水位线(watermark)到达该时间时会
被触发
         * @param time
         */
        void registerEventTimeTimer(long time);

        /**
         * 根据给定的触发时间(trigger time)来删除processing-time计时器
         * 如果这个timer不存在，那么该方法不会起作用,
         * 即该计时器(timer)之前已经被注册了，并且没有过时
         *
         * @param time
         */
        void deleteProcessingTimeTimer(long time);

        /**
         * 根据给定的触发时间(trigger time)来删除event-time 计时器
         * 如果这个timer不存在，那么该方法不会起作用,
         *       即该计时器(timer)之前已经被注册了，并且没有过时
         * @param time
         */
        void deleteEventTimeTimer(long time);
    }
```

TimerService提供了以下几种方法：

- currentProcessingTime()

返回当前的处理时间

- currentWatermark()

返回当前event-time水位线(watermark)时间戳

- registerProcessingTimeTimer(long time)

针对当前key，注册一个processing time计时器(timers)，当processing time的时间等于该计时器时钟时会被调用

- registerEventTimeTimer(long time)

针对当前key，注册一个event time计时器(timers)，当水位线时间戳大于等于该计时器时钟时会被调用

- deleteProcessingTimeTimer(long time)

针对当前key，删除一个之前注册过的processing time计时器(timers)，如果这个timer不存在，那么该方法不会起作用

- deleteEventTimeTimer(long time)

针对当前key，删除一个之前注册过的event time计时器(timers)，如果这个timer不存在，那么该方法不会起作用

当计时器触发时，会回调onTimer()函数，系统对于ProcessElement()方法和onTimer()方法的调用是同步的

注意:上面的源码中有两个Error 信息,这就说明计时器只能在keyed streams上使用，常见的用途是在某些key值不在使用后清除keyed state，或者实现一些基于时间的自定义窗口逻辑。如果要在一个非KeyedStream上使用计时器，可以使用KeySelector返回一个固定的分区值(比如返回一个常数)，这样所有的数据只会发送到一个分区。

使用案例

下面将使用Process Function的side output功能进行分流处理，具体代码如下：

```
public class ProcessFunctionExample {

    // 定义side output标签
    static final OutputTag<UserBehaviors> buyTags = new OutputTag<UserBehaviors>("buy") {
    };
    static final OutputTag<UserBehaviors> cartTags = new OutputTag<UserBehaviors>("cart") {
    };
    static final OutputTag<UserBehaviors> favTags = new OutputTag<UserBehaviors>("fav") {
    };
```

```
    static class SplitStreamFunction extends ProcessFunction<UserBehavio
rs, UserBehaviors> {

        @Override
        public void processElement(UserBehaviors value, Context ctx, Col
lector<UserBehaviors> out) throws Exception {
            switch (value.behavior) {
                case "buy":
                    ctx.output(buyTags, value);
                    break;
                case "cart":
                    ctx.output(cartTags, value);
                    break;
                case "fav":
                    ctx.output(favTags, value);
                    break;
                default:
                    out.collect(value);
            }
        }
    }
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env = StreamExecutionEnvironment.getE
xecutionEnvironment().setParallelism(1);

        // 模拟数据源[userId,behavior,product]
        SingleOutputStreamOperator<UserBehaviors> splitStream = env.from
Elements(
                new UserBehaviors(1L, "buy", "iphone"),
                new UserBehaviors(1L, "cart", "huawei"),
                new UserBehaviors(1L, "buy", "logi"),
                new UserBehaviors(1L, "fav", "oppo"),
                new UserBehaviors(2L, "buy", "huawei"),
                new UserBehaviors(2L, "buy", "onemore"),
                new UserBehaviors(2L, "fav", "iphone")).process(new Spli
tStreamFunction());

        //获取分流之后购买行为的数据
        splitStream.getSideOutput(buyTags).print("data_buy");
        //获取分流之后加购行为的数据
        splitStream.getSideOutput(cartTags).print("data_cart");
        //获取分流之后收藏行为的数据
        splitStream.getSideOutput(favTags).print("data_fav");

        env.execute("ProcessFunctionExample");
    }
}
```

总结

本文首先介绍了Flink提供的几种底层Process Function API，这些API可以访问时间戳和水位线，同时支持注册一个计时器，进行调用回调函数onTimer()。接着从源码的角度解读了这些API的共同部分，详细解释了每个方法的具体含义和使用方式。最后，给出了一个Process Function常见使用场景案例，使用其实现分流处理。除此之外，用户还可以使用这些函数，通过注册计时器，在回调函数中定义处理逻辑，使用非常的灵活。