Flink 原理与实现：Session Window

摘要： 在[上一篇文章：Window机制](http://wuchong.me/blog/2016/05/25/flink-internals-window-mechanism/)中，我们介绍了窗口的概念和底层实现，以及 Flink 一些内建的窗口，包括滑动窗口、翻滚窗口。本文将深入讲解一种较为特殊的窗口：会话窗口（session window）。建议您在阅读完上一篇文章的基础上再阅读本文。 当我们

在上一篇文章：Window机制中，我们介绍了窗口的概念和底层实现，以及 Flink 一些内建的窗口，包括滑动窗口、翻滚窗口。本文将深入讲解一种较为特殊的窗口：会话窗口（session window）。建议您在阅读完上一篇文章的基础上再阅读本文。

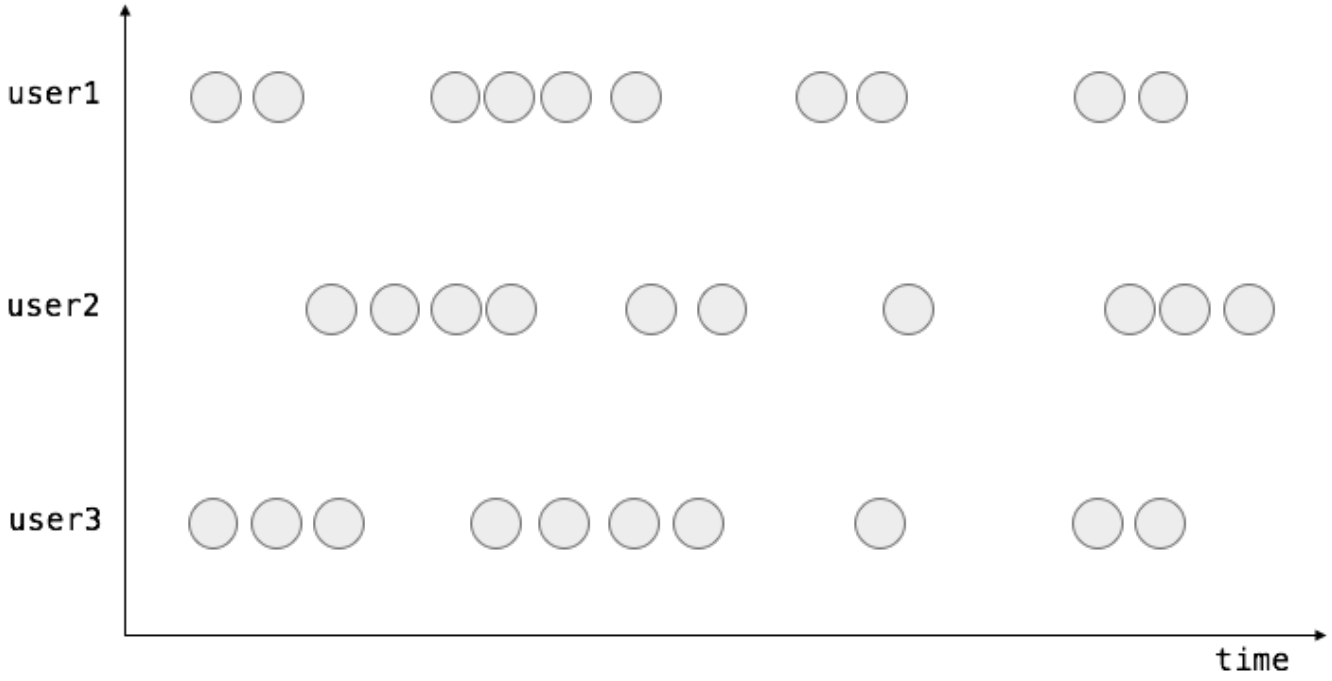
当我们需要分析用户的一段交互的行为事件时，通常的想法是将用户的事件流按照“session”来分组。session 是指一段持续活跃的期间，由活跃间隙分隔开。通俗一点说，消息之间的间隔小于超时阈值（sessionGap）的，则被分配到同一个窗口，间隔大于阈值的，则被分配到不同的窗口。目前开源领域大部分的流计算引擎都有窗口的概念，但是没有对 session window 的支持，要实现 session window，需要用户自己去做完大部分事情。而当 Flink 1.1.0 版本正式发布时，Flink 将会是开源流计算领域第一个内建支持 session window 的引擎。

在 Flink 1.1.0 之前，Flink 也可以通过自定义的window assigner和trigger来实现一个基本能用的 session window。`release-1.0` 版本中提供了一个实现 session window 的 example：SessionWindowing。这个session window范例的实现原理是，基于GlobleWindow这个 window assigner，将所有元素都分配到同一个窗口中，然后指定一个自定义的trigger来触发执行窗口。这个trigger的触发机制是，对于每个到达的元素都会根据其时间戳（timestamp）注册一个会话超时的定时器（timestamp+sessionTimeout），并移除上一次注册的定时器。最新一个元素到达后，如果超过 sessionTimeout 的时间还没有新元素到达，那么trigger就会触发，当前窗口就会是一个session window。处理完窗口后，窗口中的数据会清空，用来缓存下一个session window的数据。

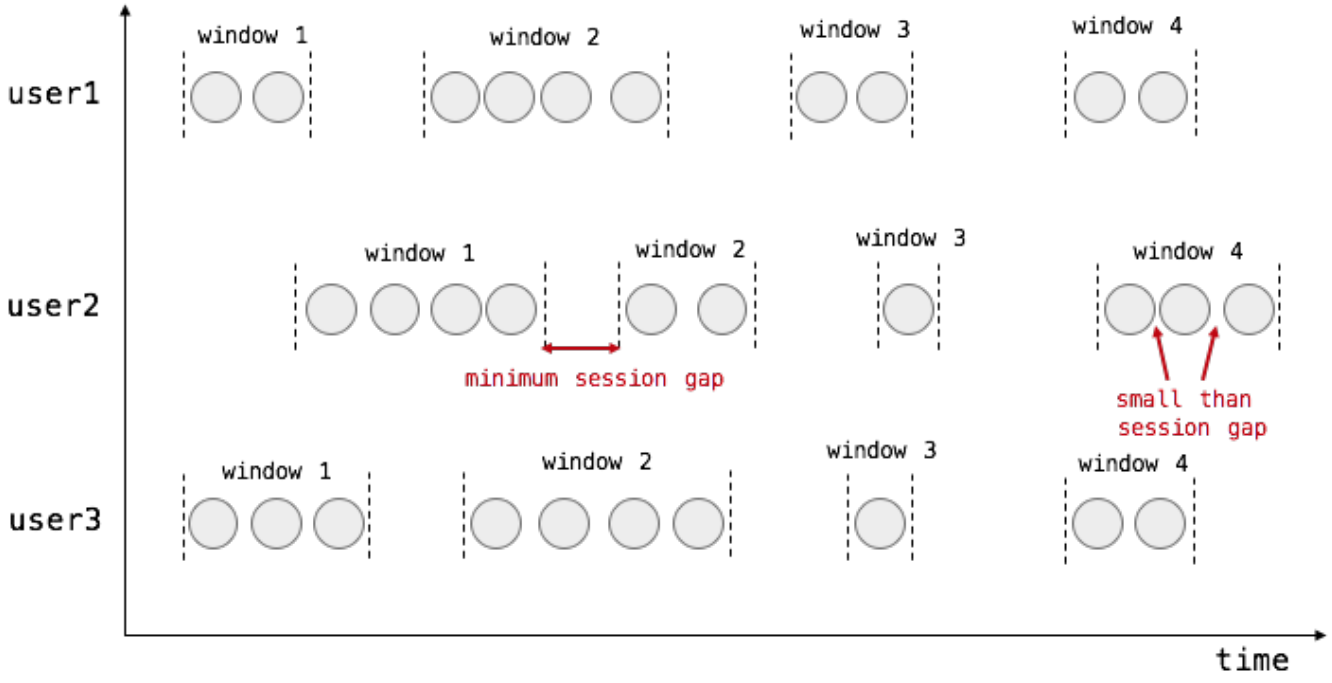
但是这种session window的实现是非常弱的，无法应用到实际生产环境中的。因为它无法处理乱序 event time 的消息。 而在即将到来的 Flink 1.1.0 版本中，Flink 提供了对 session window 的直接支持，用户可以通过`SessionWindows.withGap()`来轻松地定义 session widnow，而且能够处理乱序消息。Flink 对 session window 的支持主要借鉴自 Google 的 DataFlow 。

Session Window in Flink

假设有这么个场景，用户点开手机淘宝后会进行一系列的操作（点击、浏览、搜索、购买、切换tab等），这些操作以及对应发生的时间都会发送到服务器上进行用户行为分析。那么用户的操作行为流的样例可能会长下面这样：



通过上图，我们可以很直观地观察到，用户的行为是一段一段的，每一段内的行为都是连续紧凑的，段内行为的关联度要远大于段之间行为的关联度。我们把每一段用户行为称之为“session”，段之间的空档我们称之为“session gap”。所以，理所当然地，我们应该按照 session window 对用户的行为流进行切分，并计算每个session的结果。如下图所示：



为了定义上述的窗口切分规则，我们可以使用 Flink 提供的 `SessionWindows` 这个 widnow

assigner API。如果你用过 `SlidingEventTimeWindows`、`TumlingProcessingTimeWindows`等，你会对这个很熟悉。

```
DataStream input = …
DataStream result = input
  .keyBy(<key selector>)
  .window(SessionWindows.withGap(Time.seconds(<seconds>))
  .apply(<window function>) // or reduce() or fold()
```

这样，Flink 就会基于元素的时间戳，自动地将元素放到不同的session window中。如果两个元素的时间戳间隔小于 session gap，则会在同一个session中。如果两个元素之间的间隔大于session gap，且没有元素能够填补上这个gap，那么它们会被放到不同的session中。

底层实现

为了实现 session window，我们需要扩展 Flink 中的窗口机制，使得能够支持窗口合并。要理解其原因，我们需要先了解窗口的现状。在上一篇文章中，我们谈到了 Flink 中 WindowAssigner 负责将元素分配到哪个/哪些窗口中去，Trigger 决定了一个窗口何时能够被计算或清除。当元素被分配到窗口之后，这些窗口是固定的不会改变的，而且窗口之间不会相互作用。

对于session window来说，我们需要窗口变得更灵活。基本的思想是这样的：`SessionWindows` assigner 会为每个进入的元素分配一个窗口，该窗口以元素的时间戳作为起始点，时间戳加会话超时时间为结束点，也就是该窗口为`[timestamp, timestamp+sessionGap)`。比如我们现在到了两个元素，它们被分配到两个独立的窗口中，两个窗口目前不相交，如图：

当第三个元素进入时，分配到的窗口与现有的两个窗口发生了叠加，情况变成了这样：

由于我们支持了窗口的合并，`WindowAssigner`可以合并这些窗口。它会遍历现有的窗口，并告诉系统哪些窗口需要合并成新的窗口。Flink 会将这些窗口进行合并，合并的主要内容有两部分：

1. 需要合并的窗口的底层状态的合并（也就是窗口中缓存的数据，或者对于聚合窗口来说是一个聚合值）
2. 需要合并的窗口的Trigger的合并（比如对于EventTime来说，会删除旧窗口注册的定时器，并注册新窗口的定时器）

总之，结果是三个元素现在在同一个窗口中了：

需要注意的是，对于每一个新进入的元素，都会分配一个属于该元素的窗口，都会检查并合并现有的窗口。在触发窗口计算之前，每一次都会检查该窗口是否可以和其他窗口合并，直到trigger触发后，会将该窗口从窗口列表中移除。对于 event time 来说，窗口的触发是要等到大于窗口结束时间的 watermark 到达，当watermark没有到，窗口会一直缓存着。所以基于这种机制，可以做到对乱序消息的支持。

这里有一个优化点可以做，因为每一个新进入的元素都会创建属于该元素的窗口，然后合并。如果新元素连续不断地进来，并且新元素的窗口一直都是可以和之前的窗口重叠合并的，那么其实这里多了很多不必要的创建窗口、合并窗口的操作，我们可以直接将新元素放到那个已存在的窗口，然后扩展该窗口的大小，看起来就像和新元素的窗口合并了一样。

源码分析

FLINK-3174 这个JIRA中有对 Flink 如何支持 session window 的详细说明，以及代码更新。建议可以结合该 PR 的代码来理解本文讨论的实现原理。

为了扩展 Flink 中的窗口机制，使得能够支持窗口合并，首先 window assigner 要能合并现有的窗口，Flink 增加了一个新的抽象类 `MergingWindowAssigner` 继承自 `WindowAssigner`，这里面主要多了一个 `mergeWindows` 的方法，用来决定哪些窗口是可以合并的。

```
public abstract class MergingWindowAssigner<T, W extends Window> extends WindowAs
    private static final long serialVersionUID = 1L;

    /**
     * 决定哪些窗口需要被合并。对于每组需要合并的窗口，都会调用 callback.merge(toBeMerged,
     *
     * @param windows 现存的窗口集合 The window candidates.
     * @param callback 需要被合并的窗口会回调 callback.merge 方法
     */
    public abstract void mergeWindows(Collection<W> windows, MergeCallback<W> cal

    public interface MergeCallback<W> {

        /**
         * 用来声明合并窗口的具体动作（合并窗口底层状态、合并窗口trigger等）。
         *
         * @param toBeMerged  需要被合并的窗口列表
         * @param mergeResult 合并后的窗口
         */
        void merge(Collection<W> toBeMerged, W mergeResult);
    }
}
```

所有已经存在的 assigner 都继承自 `WindowAssigner`，只有新加入的 session window assigner 继承自 `MergingWindowAssigner`，如：`ProcessingTimeSessionWindows` 和`EventTimeSessionWindows`。

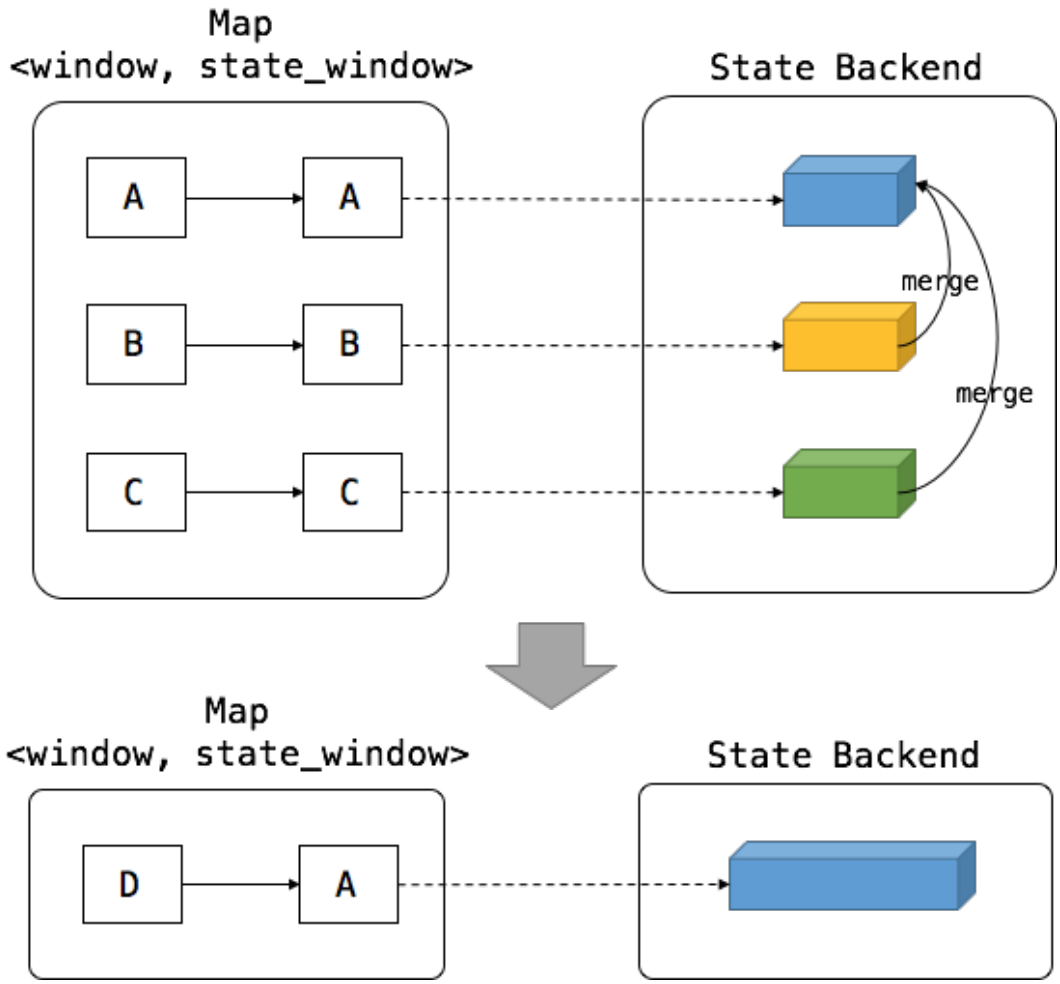
另外，Trigger 也需要能支持对合并窗口后的响应，所以 Trigger 添加了一个新的接口 `onMerge(W window, OnMergeContext ctx)`，用来响应发生窗口合并之后对trigger的相关动作，比如根据合并后的窗口注册新的 event time 定时器。

OK，接下来我们看下最核心的代码，也就是对于每个进入的元素的处理，代码位于`WindowOperator.processElement`方法中，如下所示：

```
public void processElement(StreamRecord<IN> element) throws Exception {
    Collection<W> elementWindows = windowAssigner.assignWindows(element.getValue(
    final K key = (K) getStateBackend().getCurrentKey();
    if (windowAssigner instanceof MergingWindowAssigner) {
        // 对于session window 的特殊处理，我们只关注该条件块内的代码
        MergingWindowSet<W> mergingWindows = getMergingWindowSet();

        for (W window: elementWindows) {
            final Tuple1<TriggerResult> mergeTriggerResult = new Tuple1<>(Trigger

            // 加入新窗口，如果没有合并发生,那么actualWindow就是新加入的窗口
            // 如果有合并发生，那么返回的actualWindow即为合并后的窗口，
            // 并且会调用 MergeFunction.merge 方法，这里方法中的内容主要是更新trigger，合
            W actualWindow = mergingWindows.addWindow(window, new MergingWindowSe
                @Override
                public void merge(W mergeResult,
                        Collection<W> mergedWindows, W stateWindowResult,
                        Collection<W> mergedStateWindows) throws Exception {
                    context.key = key;
                    context.window = mergeResult;

                    // 这里面会根据新窗口的结束时间注册新的定时器
                    mergeTriggerResult.f0 = context.onMerge(mergedWindows);

                    // 删除旧窗口注册的定时器
                    for (W m: mergedWindows) {
                        context.window = m;
                        context.clear();
                    }

                    // 合并旧窗口(mergedStateWindows)中的状态到新窗口（stateWindowResu
                    getStateBackend().mergePartitionedStates(stateWindowResult,
                            mergedStateWindows,
                            windowSerializer,
                            (StateDescriptor<? extends MergingState<?,?>, ?>) win
                }
            });

            // 取 actualWindow 对应的用来存状态的窗口
```

```
                W stateWindow = mergingWindows.getStateWindow(actualWindow);
                // 从状态后端拿出对应的状态
                AppendingState<IN, ACC> windowState = getPartitionedState(stateWindow
                // 将新进入的元素数据加入到新窗口（或者说合并后的窗口）中对应的状态中
                windowState.add(element.getValue());

                context.key = key;
                context.window = actualWindow;

                // 检查是否需要fire or purge
                TriggerResult triggerResult = context.onElement(element);

                TriggerResult combinedTriggerResult = TriggerResult.merge(triggerResu

                // 根据trigger结果决定怎么处理窗口中的数据
                processTriggerResult(combinedTriggerResult, actualWindow);
            }

        } else {
            // 对于普通window assigner的处理，  这里我们不关注
            for (W window: elementWindows) {

                AppendingState<IN, ACC> windowState = getPartitionedState(window, win
                        windowStateDescriptor);

                windowState.add(element.getValue());

                context.key = key;
                context.window = window;
                TriggerResult triggerResult = context.onElement(element);

                processTriggerResult(triggerResult, window);
            }
        }
    }
```

其实这段代码写的并不是很clean，并且不是很好理解。在第六行中有用到`MergingWindowSet`，这个类很重要所以我们先介绍它。这是一个用来跟踪窗口合并的类。比如我们有A、B、C三个窗口需要合并，合并后的窗口为D窗口。这三个窗口在底层都有对应的状态集合，为了避免代价高昂的状态替换（创建新状态是很昂贵的），我们保持其中一个窗口作为原始的状态窗口，其他几个窗口的数据合并到该状态窗口中去，比如随机选择B作为状态窗口，那么A和C窗口中的数据需要合并到B窗口中去。这样就没有新状态产生了，但是我们需要额外维护窗口与状态窗口之间的映射关系（D->B），这就是`MergingWindowSet`负责的工作。这个映射关系需要在失败重启后能够恢复，所以`MergingWindowSet`内部也是对该映射关系做了容错。状态合并的工作示意图如下所示：



然后我们来解释下processElement的代码，首先根据window assigner为新进入的元素分配窗口集合。接着进入第一个条件块，取出当前的`MergingWindowSet`。对于每个分配到的窗口，我们就会将其加入到`MergingWindowSet`中（`addWindow`方法），由`MergingWindowSet`维护窗口与状态窗口之间的关系，并在需要窗口合并的时候，合并状态和trigger。然后根据映射关系，取出结果窗口对应的状态窗口，根据状态窗口取出对应的状态。将新进入的元素数据加入到该状态中。最后，根据trigger结果来对窗口数据进行处理，对于session window来说，这里都是不进行任何处理的。真正对窗口处理是由定时器超时后对完成的窗口调用`processTriggerResult`。

总结

本文在上一篇文章：Window机制的基础上，深入讲解了 Flink 是如何支持 session window 的，核心的原理是窗口的合并。Flink 对于 session window 的支持很大程度上受到了 Google DataFlow 的启发，所以也建议阅读下 DataFlow 的论文。

参考资料

- Introduction to Flink Streaming - Part 7 : Implementing Session Windows using Custom Trigger
- [How Apache Flink Enables New Streaming Applications

Part III: Session Windowing in Flink](http://data-artisans.com/session-windowing-in-flink/)

- Google DataFlow paper
- Google DataFlow Document
- FLINK-3174: Add merging WindowAssigner