**1.stateless transformation**

无状态的转换算子：流处理器不涉及状态的处理和存储

**1.1 branch**

分支 ：将一个stream转换为1到多个Stream stream----->stream[]

//branch 分流
KStream<String, String>[] streams = kStream.branch((k, v) -> v.startsWith("A"), (k, v) -> v.startsWith("B"), (k, v) -> true);
streams[0].foreach((k,v)-> System.out.println(k+" |" +v)); //遍历以A开头
streams[1].foreach((k,v)-> System.out.println(k+": "+v)); //遍历以B开头
streams[2].foreach((k,v)-> System.out.println(k+"||"+v)); //遍历其他

**1.2 filter**

过滤:将一个Stream经过boolean函数处理，保留符合条件的结果

//filter 过滤 保留record value为Hello开头的结果
kStream.filter((k,v) -> v.startsWith("Hello")).foreach((k,v) -> System.out.println(k+"\t"+v));

**1.3 filterNot**

翻转过滤:将一个Stream经过Boolean函数处理保留不符合条件的结果

//翻转过滤 保留不以Hello开头
KStream<String, String> stream = kStream.filterNot((k, v) -> v.startsWith("Hello"));
stream.foreach((k,v)-> System.out.println(k+" :"+v));

**1.4 flatMap**

将一个record展开，产生0到多个record record—>record1,record2…

//flatMap展开
kStream.flatMap((k,v)->{
List<KeyValue<String, String>> keyValues = new ArrayList<>();
String[] words = v.split(" ");
for (String word : words) {
keyValues.add(new KeyValue<String, String>(k,word));
}
return keyValues;
}).foreach((k,v)-> System.out.println(k+" | "+v));

**1.5 flatMapValues**

将一条record变成多条record并且将多条记录展开

（k,v）–>(k,v1),(k,v2)…

//flatMapValues
kStream.flatMapValues((v)-> Arrays.asList(v.split(" "))).foreach((k,v)-> System.out.println(k+" | "+v));

**1.6 foreach**

终止操作，为每一个record提供一种无状态的操作

.foreach((k,v) -> System.out.println(k+"\t"+v));

**1.7 GroupByKey | GroupBy**

GroupByKey:根据key进行分组

GroupBy:根据自定义的信息进行分组

kStream
.flatMap((k, v) -> {
String[] words = v.split(" ");
List<KeyValue<String, String>> keyValues = new ArrayList<>();
for (String word : words) {
keyValues.add(new KeyValue<String, String>(word, word));
}
return keyValues;
})
.groupByKey()
.count()
.toStream()
.print(Printed.toSysOut()); //标准输出样式

**1.8 map | mapValues**

将一条record映射为另外一条record

kStream.map((k,v) -> new KeyValue<String,Long>(k,(long) v.length())).foreach((k,v) -> System.out.println(k +"\t"+v));

**1.9 Merge**

将两个流合并为一个

KStream<byte[], String> stream1 = ...;
KStream<byte[], String> stream2 = ...;
KStream<byte[], String> merged = stream1.merge(stream2);

**1.10 Peek**

作为程序执行的探针，一般用于debug调试，因为peek并不会对后续的流数据带来任何影响。

KStream<byte[], String> unmodifiedStream = stream.peek((key, value) -> System.out.println("key=" + key + ", value=" + value));

**1.11 Print**

最终操作，将每一个record进行输出打印

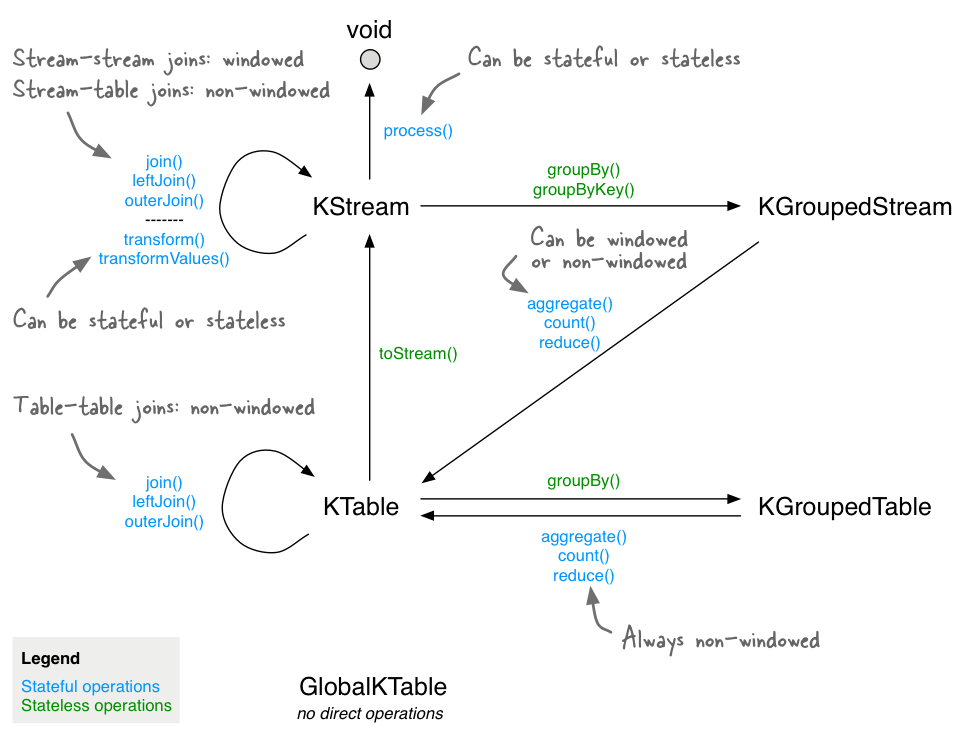
stream.print(Printed.toSysOut());
stream.print(Printed.toFile("streams.out").withLabel("streams"));

修改记录中key(k,v)---->(newkey ,v)

KStream<String, String> rekeyed = stream.selectKey((key, value) -> value.split(" ")[0])

**2.statful transformation**

有状态的转换算子，处理器【Processor】在进行处理时需要更新状态或者从历史状态中恢复数据



**2.1 Aggregate**

聚合 有状态的转换算子

KTable<String, Long> kTable = kStream
.flatMapValues(value -> Arrays.asList(value.split(" ")))
.groupBy((k, v) -> v)
// 第一参数：聚合的初始值 第二参数：聚合逻辑 第三个参数：【必须】指定状态存储的KV数据类型
.aggregate(
()-> 0L,
(k,v,agg) -> 1L+agg,
Materialized.<String,Long, KeyValueStore<Bytes,byte[]>>as("c152")
.withKeySerde(Serdes.String())
.withValueSerde(Serdes.Long()));
**2.2 Count**

统计key相同的record出现的次数

// 指定状态存储的k v的结构类型
.count(Materialized.<String, Long, KeyValueStore<Bytes,byte[]>>as("c158").withKeySerde(Serdes.String()).withValueSerde(Serdes.Long()));

**2.3 Reduce**

规约 计算 有状态的转换算子

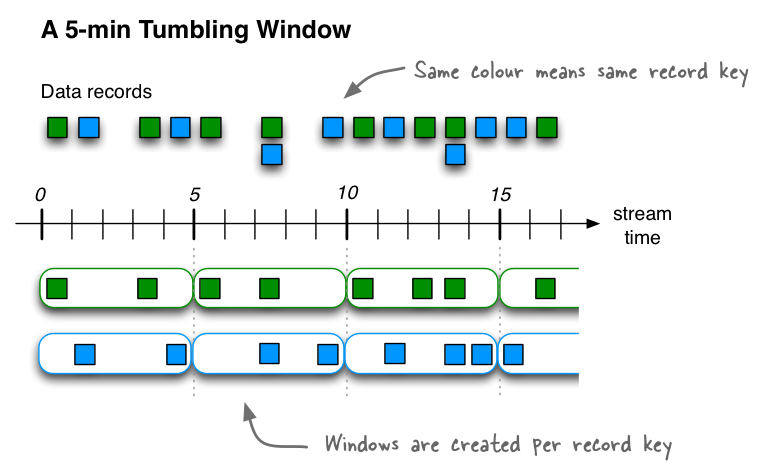
//Reducer
KTable<String, Long> kTable = kStream
.flatMapValues(value ->
Arrays.asList(value.split(" ")))
.map((String k, String v) -> new KeyValue<String, Long>(v, 1L))
.groupByKey(Grouped.with(Serdes.String(), Serdes.Long()))
.reduce((v1, v2) -> v1 + v2, Materialized.<String, Long, KeyValueStore<Bytes, byte[]>>as("c152")
.withKeySerde(Serdes.String())
.withValueSerde(Serdes.Long()));

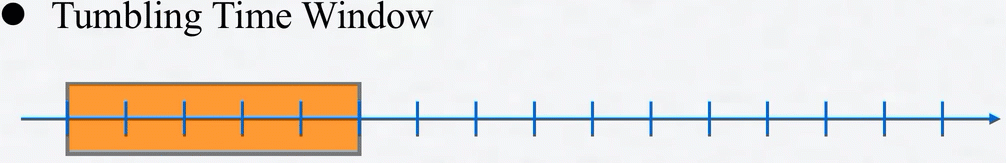
**3.窗口操作**

micro batch(微批)，时间维度数据范围的计算

**3.1 Tumbling（翻滚）固定大小 无重叠**

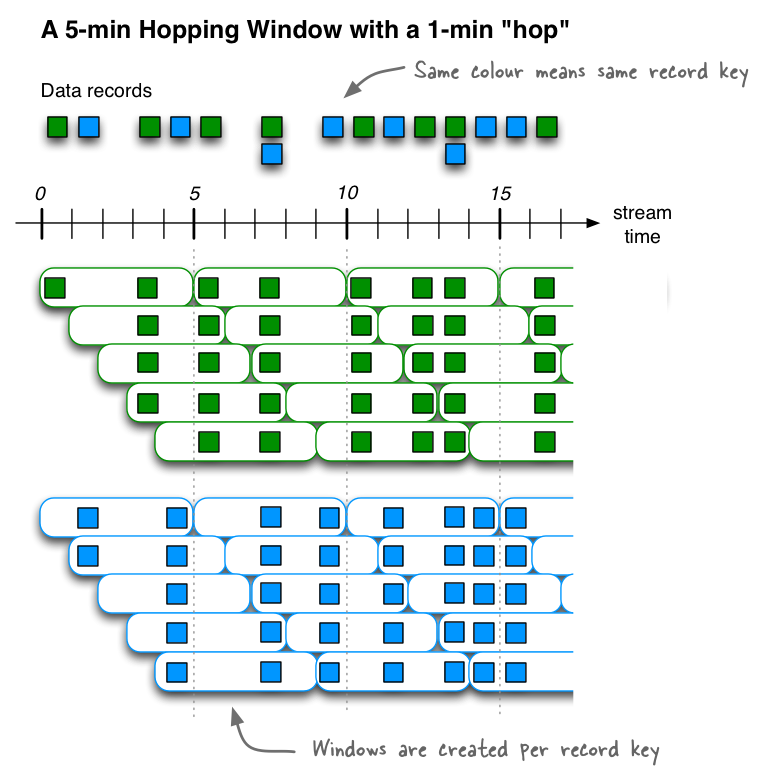
翻滚窗口将流元素按照固定的时间间隔，拆分成指定的窗口，窗口和窗口间元素之间没有重叠。在下图不同颜色的record表示不同的key。可以看是在时间窗口内，每个key对应一个窗口。前闭后开

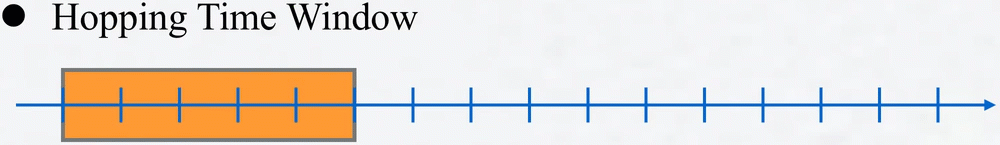




//===========================翻滚窗口=================================
kStream
.flatMapValues(value -> Arrays.asList(value.split(" ")))
.groupBy((k, v) -> v)
.windowedBy(TimeWindows.of(Duration.ofSeconds(10)))
.count(Materialized.<String, Long, WindowStore<Bytes,byte[]>>as("c152").withKeySerde(Serdes.String()).withValueSerde(Serdes.Long()));

**3.2 Hopping (跳跃) 固定大小 有重叠**





//=================================跳跃窗口==========================================
KTable<Windowed<String>, Long> kTable = kStream
.flatMapValues(value -> Arrays.asList(value.split(" ")))
.groupBy((k, v) -> v)
// 将分组后的数据按照窗口进行划分
// 翻滚窗口 时间间隔10s
// 第一个窗口：now:0 - 10s 计算
// 第二个窗口：5-15 计算 (5-10)归属于第一个和第二个窗口
// 10-20
// ...
.windowedBy(TimeWindows.of(Duration.ofSeconds(10)).advanceBy(Duration.ofSeconds(5)))
// 指定状态存储的k v的结构类型
.count(Materialized.<String, Long, WindowStore<Bytes,byte[]>>as("c152").withKeySerde(Serdes.String()).withValueSerde(Serdes.Long()));
//===========================================================================

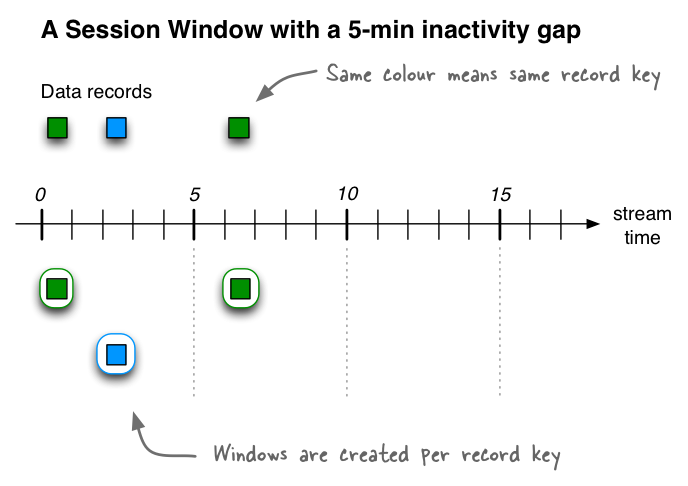
**3.3 session window**

Session 窗口的大小动态 无重叠 数据驱动的窗口

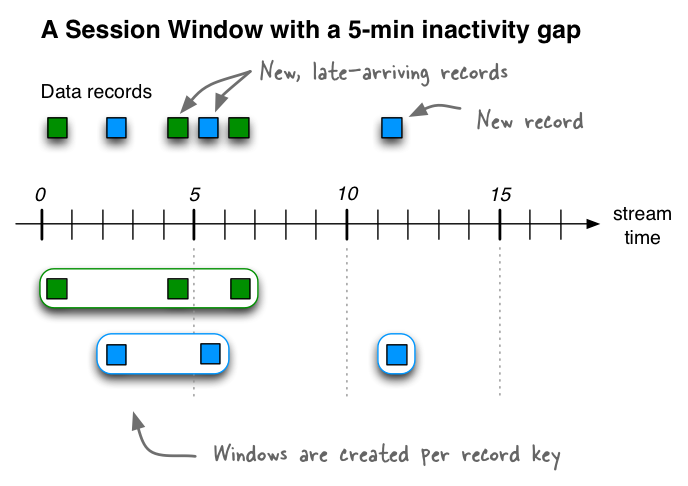
回顾：Servelt Session 会话对象，一旦使用Session，会话会自动延长30min，Session超时策略（服务器自动删除30min未使用的Session）

Session Window该窗口用于对Key做Group后的聚合操作中。它需要对Key做分组，然后对组内的数据根据业务需求定义一个窗口的起始点和结束点。一个典型的案例是，希望通过Session Window计算某个用户访问网站的时间。对于一个特定的用户（用Key表示）而言，当发生登录操作时，该用户（Key）的窗口即开始，当发生退出操作或者超时时，该用户（Key）的窗口即结束。窗口结束时，可计算该用户的访问时间或者点击次数等。

Session Windows用于将基于key的事件聚合到所谓的会话中，其过程称为session化。会话表示由定义的不活动间隔（或“空闲”）分隔的活动时段。处理的任何事件都处于任何现有会话的不活动间隙内，并合并到现有会话中。如果事件超出会话间隙，则将创建新会话。会话窗口的主要应用领域是用户行为分析。基于会话的分析可以包括简单的指标.



如果我们接收到另外三条记录（包括两条迟到的记录），那么绿色记录key的两个现有会话将合并为一个会话，从时间0开始到结束时间6，包括共有三条记录。蓝色记录key的现有会话将延长到时间5结束，共包含两个记录。最后，将在11时开始和结束蓝键的新会话。



//==================================会话窗口=========================================
KTable<Windowed<String>, Long> kTable = kStream
.flatMapValues(value -> Arrays.asList(value.split(" ")))
.groupBy((k, v) -> v)
// 将分组后的数据按照窗口进行划分
// 翻滚窗口 时间间隔10s
// 第一个窗口：now:0 - 10s 计算
// 第二个窗口：5-15 计算 (5-10)归属于第一个和第二个窗口
// 10-20
// ...
.windowedBy(SessionWindows.with(Duration.ofSeconds(10)))
// 指定状态存储的k v的结构类型
.count(Materialized.<String, Long, SessionStore<Bytes, byte[]>>as("CC").withKeySerde(Serdes.String()).withValueSerde(Serdes.Long()));
//===========================================================================
kTable.toStream().foreach((k, v) -> { // 窗口计算指的是对窗口内的数据进行计算
long start = k.window().start();
long end = k.window().end();
SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
String d1 = sdf.format(new Date(start));
String d2 = sdf.format(new Date(end));
System.out.println(d1 + "\t" + d2 + "\t" + k.key() + "\t" + v);
});