## 2.1 window源码分析

### 2.1.1 Window

```
public abstract class Window {

    /**
     * Gets the largest timestamp that still belongs to this window.
     *
     * @return The largest timestamp that still belongs to this window.
     */
    public abstract long maxTimestamp();
}
```

可以看出，**Window**抽象类仅有一个**maxTimestamp()**方法用于获取仍属于该窗口的最大时间戳。下面的全局窗口和时间窗口都是从该类扩展的。

### 2.1.2 Global window

```
public class GlobalWindow extends Window {

    private static final GlobalWindow INSTANCE = new GlobalWindow();

    private GlobalWindow() { }

    public static GlobalWindow get()
        { return INSTANCE;
    }

    @Override
    public long maxTimestamp()
        { return Long.MAX_VALUE;
    }

    @Override
    public boolean equals(Object o) {
        return this == o || !(o == null || getClass() !=o.getClass());
    }

    @Override
    public int hashCode() { return
        0;
```

```
    }

    @Override
    public String toString()
        { return
        "GlobalWindow";
    }

    /**
     * A {@link TypeSerializer} for {@link GlobalWindow}.
     */
    public static class Serializer extends TypeSerializerSingleton<GlobalWindow>
{
        private static final long serialVersionUID = 1L;

        @Override
        public boolean isImmutableType()
            { return true;
        }

        @Override
        public GlobalWindow createInstance()
            { return GlobalWindow.INSTANCE;
        }

        @Override
        public GlobalWindow copy(GlobalWindow from)
            { return from;
        }

        @Override
        public GlobalWindow copy(GlobalWindow from, GlobalWindow reuse)
            { return from;
        }

        @Override
        public int getLength()
            { return 0;
        }

        @Override
        public void serialize(GlobalWindow record, DataOutputView target) throws
IOException {
            target.writeByte(0);
        }

        @Override
        public GlobalWindow deserialize(DataInputView source) throws IOException
{
            source.readByte();
            return GlobalWindow.INSTANCE;
        }

        @Override
        public GlobalWindow deserialize(GlobalWindow reuse,
                DataInputView source) throws IOException {
            source.readByte();
            return GlobalWindow.INSTANCE;
        }
```

```
        @Override
        public void copy(DataInputView source, DataOutputView target) throws
IOException {
            source.readByte();
            target.writeByte(0);
        }

        // ------------------------------------------------------------------------
---

        @Override
        public TypeSerializerSnapshot<GlobalWindow> snapshotConfiguration()
            { return new GlobalWindowSerializerSnapshot();
        }

        /**
         * Serializer configuration snapshot for compatibility and format
evolution.
         */
        @SuppressWarnings("WeakerAccess")
        public static final class GlobalWindowSerializerSnapshot extends
SimpleTypeSerializerSnapshot<GlobalWindow> {

            public GlobalWindowSerializerSnapshot()
                { super(GlobalWindow.Serializer::new)
                ;
            }
        }
    }
}
```

**GlobalWindow**提供了**get()**静态方法用于获取**GlobalWindow**实例，**maxTimestamp()**统一返回**Long**的最大值，而**hashCode**统一返回**0**。

### 2.1.3 Time window

```
public class TimeWindow extends Window {

    private final long start;
    private final long end;

    public TimeWindow(long start, long end)
        { this.start = start;
        this.end = end;
    }

    /**
     * Gets the starting timestamp of the window. This is the first timestamp that
belongs
     * to this window.
     *
     * @return The starting timestamp of this window.
     */
    public long getStart() {
```

```
        return start;
    }

    /**
     * Gets the end timestamp of this window. The end timestamp is exclusive,
meaning it
     * is the first timestamp that does not belong to this window any more.
     *
     * @return The exclusive end timestamp of this window.
     */
    public long getEnd()
        { return end;
    }

    /**
     * Gets the largest timestamp that still belongs to this window.
     *
     * <p>This timestamp is identical to {@code getEnd() - 1}.
     *
     * @return The largest timestamp that still belongs to this window.
     *
     * @see #getEnd()
     */
    @Override
    public long maxTimestamp()
        { return end - 1;
    }

    @Override
    public boolean equals(Object o)
        { if (this == o) {
            return true;
        }
        if (o == null || getClass() != o.getClass())
            { return false;
        }

        TimeWindow window = (TimeWindow) o;

        return end == window.end && start == window.start;
    }

    @Override
    public int hashCode() {
        return MathUtils.longToIntWithBitMixing(start + end);
    }

    @Override
    public String toString()
        { return "TimeWindow{" +
                "start=" + start + ",
                end=" + end + '}';
    }

    /**
     * Returns {@code true} if this window intersects the given window
     * or if this window is just after or before the given window.
```

```
     */
    public boolean intersects(TimeWindow other) {
        return this.start <= other.end && this.end >= other.start;
    }

    /**
     * Returns the minimal window covers both this window and the given window.
     */
    public TimeWindow cover(TimeWindow other) {
        return new TimeWindow(Math.min(start, other.start), Math.max(end,
other.end));
    }

    // ------------------------------------------------------------------------
    // Serializer
    // ------------------------------------------------------------------------

    /**
     * The serializer used to write the TimeWindow type.
     */
    public static class Serializer extends TypeSerializerSingleton<TimeWindow>
        { private static final long serialVersionUID = 1L;

        @Override
        public boolean isImmutableType()
            { return true;
        }

        @Override
        public TimeWindow createInstance()
            { return null;
        }

        @Override
        public TimeWindow copy(TimeWindow from)
            { return from;
        }

        @Override
        public TimeWindow copy(TimeWindow from, TimeWindow reuse)
            { return from;
        }

        @Override
        public int getLength()
            { return 0;
        }

        @Override
        public void serialize(TimeWindow record, DataOutputView target) throws
IOException {
            target.writeLong(record.start);
            target.writeLong(record.end);
        }

        @Override
        public TimeWindow deserialize(DataInputView source) throws IOException
            { long start = source.readLong();
```

```
            long end = source.readLong();
            return new TimeWindow(start, end);
        }

        @Override
        public TimeWindow deserialize(TimeWindow reuse, DataInputView source)
throws IOException {
            return deserialize(source);
        }

        @Override
        public void copy(DataInputView source, DataOutputView target) throws
IOException {
            target.writeLong(source.readLong());
            target.writeLong(source.readLong());
        }

        // ------------------------------------------------------------------------
---

        @Override
        public TypeSerializerSnapshot<TimeWindow> snapshotConfiguration()
            { return new TimeWindowSerializerSnapshot();
```

该窗口主要用于实现时间驱动的相关操作。

可以看到。**TimeWindow**由**start**和**end2**个时间戳组成，最大时间戳为**end-1**，同时，**TimeWindow**提供了**getWindowStartWithOffset**静态方法，用于获取时间戳所属时间窗的起点，其中的**offset**为偏移量。 例如，没有偏移量的话，小时滚动窗口将按时间纪元来对齐，也就是

**1:00:00–1:59:59,2:00:00–2:59:59**等，如果你指定了**15**分钟的偏移，你将得到

**1:15:00–2:14:59,2:15:00–3:14:59**等。时间偏移主要用于调准非**0**时区的窗口，例如**:**在中国你需要指定**8**小时的时间偏移。

**intersects**方法用于判断**2**个时间窗是否有交集，**cover**方法用于求**2**个时间窗的合集，**mergeWindows**用于将时间窗集合进行合并，该方法是实现**Session Window**的关键。