Flink 状态详解

了解状态

> While many operations in a dataflow simply look at one individual event at a time (for example an event parser), some operations remember information across multiple events (for example window operators). These operations are called stateful. 虽然许多数据流仅仅是简单的处理一次，但是例如窗口聚合等等算子需要记录多个数据的信息，这些算子叫做富有状态的。

状态指用于保存中间计算结果或者缓存数据，是否需要保存这些数据，数据流又可以分为有状态的和无状态的。

简单来说，不依赖于上下游数据的数据流就是无状态数据流，反之则是有状态的。

状态的应用场景

如上文所说，依赖上下游数据的数据流就需要使用到状态，例如：

- 求和
- 去重
- CEP

等等。

```
public static void main(String[] args) throws Exception {
    StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
    env.setParallelism(1);

    env.fromElements(1, 2, 3, 3).keyBy(i -> 0).map(new RichMapFunction<Integer,
Integer>() {
        private ValueState<Integer> valueState;

        @Override
        public Integer map(Integer value) throws Exception {
            // read from state
            int i = valueState.value() == null ? 0 : valueState.value();

            if (i > 5) {
                System.out.println("i = " + i);
            }

            // update state
            valueState.update(i + value);

            return value;
        }

        @Override
        public void open(Configuration parameters) throws Exception {
```

```
            ValueStateDescriptor<Integer> descriptor
                = new ValueStateDescriptor<Integer>("sum",
TypeInformation.of(Integer.class));
            valueState = getRuntimeContext().getState(descriptor);
        }
    }).print();

    env.execute();
}
```

以上代码输出：

```
1
2
3
i = 6
3
```

状态组件

状态存储 State store

随着状态被分为算子状态和键值状态，也存在相应的算子状态存储OperatorStateStore和键值状态存储keyedStateStore。 StateStore和StateBackend的关系为：Store是Backend的上层抽象。

其中，OperatorStateStore将状态存储在内存中：

```
public class DefaultOperatorStateBackend {
    // 状态名称到状态的映射
    private final Map<String, PartitionableListState<?>> registeredOperatorStates;
    // 状态名称到广播状态的映射
    private final Map<String, BackendWritableBroadcastState<?, ?>>
registeredBroadcastStates;
    // 已被调用的状态的映射，某个名称第一次获取时，被加入该map
    private final Map<String, PartitionableListState<?>> accessedStatesByName;
    // 已被调用的广播状态的映射，某个名称第一次获取时，被加入该map
    private final Map<String, BackendWritableBroadcastState<?, ?>>
accessedBroadcastStatesByName;
}
```

对应的，KeyedStateStore依赖于KeyedStateBackend的实现

状态后端 State Backends

当启用检查点时，状态会持久化在指定位置，防止数据丢失并进行恢复。 状态如何展示，以及在检查点位置的持久化方式取决于所选的状态后端。

例如，HashMapStateBackend会将运行中的状态存储在TaskManager中，轻量级。

目前，flink有三个默认实现的状态后端：

- MemoryStateBackend 已过期，需要使用HashMapStateBackend或JobManagerCheckpointStorage来替代
- FsStateBackend 已过期，需要使用HashMapStateBackend或FileSystemCheckpointStorage来替代
- RocksDBStateBackend(需要外部依赖)

HashMapStateBackend

> This state backend holds the working state in the memory (JVM heap) of the TaskManagers and checkpoints based on the configured.
> 将状态保存在内存中。

由于内存大小限制以及其较高的IO速度，一般用于本地调试。

JobManagerCheckpointStorage

> The checkpoints state directly to the JobManager's memory (hence the name), but savepoints will be persisted to a file system.
> 将检查点信息保存在jobmanager的内存中，保存点则是存储在文件中。

由于内存大小限制以及其较高的IO速度，一般用于本地调试。

FileSystemCheckpointStorage

> FileSystemCheckpointStorage checkpoints state as files to a file system. Each checkpoint individually will store all its files in a subdirectory that includes the checkpoint number, such as {@code hdfs://namenode:port/flink-checkpoints/chk-17/}. 将检查点状态存储在文件系统中。

比较广泛应用的状态后端。其原理是在内存中维护状态，每次检查点写入文件系统，理论上会出现OOM。

RocksDBStateBackend

类似FsStateBackend，但是状态直接写入文件系统，如果配置合理，理论上不会出现OOM，但是理论性能不如FsStateBackend。

需要在pom中新入：

```
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-statebackend-rocksdb_2.12</artifactId>
            <version>1.14.2</version>
        </dependency>
```

状态描述 StateDescriptor

用于在有状态的算子中创建分区状态。 状态既然暴露给开发者使用，那么就存在一些属性需要指定，例如名称、序列化器等，StateDescriptor用于存储这些属性。 StateDescriptor从状态后端中获取状态实例，状态后端中有则返回，无则创建。

```
public abstract class StateDescriptor<S extends State, T> {
    // 唯一标识符
    protected final String name;
    // 序列化器
    private final AtomicReference<TypeSerializer<T>> serializerAtomicReference;
    // 仅用于指定数据类型
    private TypeInformation<T> typeInfo;
    // 查询用名称
    private String queryableStateName;
    // 状态过期策略
    private StateTtlConfig ttlConfig;
    // 默认值
    protected transient T defaultValue;
}
```

State接口

```
public interface State {
    void clear();
}

// 内部状态使用的接口，相对于State，新增更多的状态操作
// 注意，不同版本内部状态差距较大，不建议直接使用
public interface InternalKvState<K, N, V> extends State {
    // 内部状态可以直接获取序列化器
    TypeSerializer<K> getKeySerializer();
    TypeSerializer<N> getNamespaceSerializer();
    TypeSerializer<V> getValueSerializer();
}
```

状态类型

> 1.14版本的Flink，全部的状态类型均在org.apache.flink.api.common.state下。

ValueState

针对单一值的状态，需要KeyedStream。

```
public interface ValueState<T> {
    // 返回状态当前的值，若当前的值为null，会返回用户指定的默认值
    // 指定默认值的方法已被弃用，系统指定默认值为null
    T value() throws IOException;
    // 更新状态的值，若更新一个null，则下次获取值时，会返回用户指定的默认值
    void update(T value) throws IOException;
}
```

一个实现类：

```
class BatchExecutionKeyValueState<K, N, T> {
    @Override
    public T value() {
        // 执行父类AbstractBatchExecutionKeyState的函数
        return getOrDefault();
    }

    @Override
    public void update(T value) {
        // 执行父类AbstractBatchExecutionKeyState的函数
        setCurrentNamespaceValue(value);
    }
}

abstract class AbstractBatchExecutionKeyState<K, N, V> {
    V getOrDefault() {
        if (currentNamespaceValue == null && defaultValue != null) {
            // 此处执行一个深拷贝，序列化器会根据泛型挑选
            // 序列化类型在声明状态时被指定
            return stateTypeSerializer.copy(defaultValue);
        }
        return currentNamespaceValue;
    }

    public void setCurrentNamespaceValue(V currentNamespaceValue) {
        this.currentNamespaceValue = currentNamespaceValue;
    }
}
```

ListState

常见的状态中，唯一一个支持非keyedStream的状态。顾名思义，是一个列表。

```
public interface ListState<T> {
    // 类似ValueState#update
    // 当传入一个empty list，状态值将会置为null
    void update(List<T> values) throws Exception;
    // 当传入一个empty list，状态值不会改变
    void addAll(List<T> values) throws Exception;
    // 来自AppendingState接口，OUT指的是类型为T的迭代器
    // 当列表为empty，将会返回null
    OUT get() throws Exception;
    // 来自AppendingState接口，OUT指的是类型为T的迭代器
    // 当传入一个null，则列表不会变化
    void add(IN value) throws Exception;
}
```

一个实现类：

```
class BatchExecutionKeyListState<K, N, T> {
    @Override
    public void update(List<T> values) {
        // 禁止传入null
        checkNotNull(values);
        // 清空当前状态列表
        clear();
        for (T value : values) {
            add(value);
        }
    }

    @Override
    public void addAll(List<T> values) {
        // 当传入一个empty时，直接返回
        if (checkNotNull(values).isEmpty()) {
            return;
        }
        for (T value : values) {
            add(value);
        }
    }

    @Override
    public Iterable<T> get() throws Exception {
        return getCurrentNamespaceValue();
    }

    @Override
    public void add(T value) {
        checkNotNull(value);
        // 当状态为null时，初始化一个empty list
        initIfNull();
        getCurrentNamespaceValue().add(value);
    }
}
```

ReducingState

仅能被KeyedStream使用，用户自定义function，每次调用add方法添加值时，最后合并为一个状态值。

```
public interface ReducingState<T> {
    // 来自AppendingState接口，OUT指的T
    OUT get() throws Exception;
    // 来自AppendingState接口，OUT指的T
    void add(IN value) throws Exception;
}
```

```
public interface ReduceFunction<T> {
    T reduce(T value1, T value2) throws Exception;
}
```

一个实现类：

```
class BatchExecutionKeyReducingState<K, N, T> {
    @Override
    public T get() {
        // 略
        return getOrDefault();
    }

    @Override
    public void add(T value) throws IOException {
        if (value == null) {
            // 当传入为null时，直接清空
            clear();
            return;
        }

        try {
            T currentNamespaceValue = getCurrentNamespaceValue();
            if (currentNamespaceValue != null) {

setCurrentNamespaceValue(reduceFunction.reduce(currentNamespaceValue, value));
            } else {
                setCurrentNamespaceValue(value);
            }
        } catch (Exception e) {
            throw new IOException("Exception while applying ReduceFunction in
reducing state", e);
        }
    }
}
```

AggregatingState

类似ReducingState，区别在于AggregatingState可以聚合两种不同类型的对象，即输入输出类型可以不同。

```
public interface AggregatingState<IN, OUT> {
    // 来自AppendingState接口
    OUT get() throws Exception;
    // 来自AppendingState接口
    void add(IN value) throws Exception;
}

public interface AggregateFunction<IN, ACC, OUT> {
    // 初始化累加器，返回一个初始化的ACC
```

```
    ACC createAccumulator();
    // 如何将IN添加进ACC
    ACC add(IN value, ACC accumulator);
    // 如果将ACC转化为OUT
    OUT getResult(ACC accumulator);
    // 如何将两个累加器累加，应用于算子减少并行度时
    ACC merge(ACC a, ACC b);
}
```

一个实现类：

```
class BatchExecutionKeyAggregatingState<K, N, IN, ACC, OUT> {
    @Override
    public OUT get() {
        // 简单来说，传入类型为IN，存储类型为ACC，输出类型为OUT
        ACC acc = getOrDefault();
        // 从ACC到OUT的步骤需要实现getResult方法
        return acc != null ? aggFunction.getResult(acc) : null;
    }

    @Override
    public void add(IN value) throws IOException {
        // 传入null时，直接清空
        if (value == null) {
            clear();
            return;
        }

        try {
            if (getCurrentNamespaceValue() == null) {
                setCurrentNamespaceValue(aggFunction.createAccumulator());
            }
            setCurrentNamespaceValue(aggFunction.add(value,
getCurrentNamespaceValue()));
        } catch (Exception e) {
            throw new IOException(
                    "Exception while applying AggregateFunction in aggregating
state", e);
        }
    }
}
```

MapState

仅KeyedStream使用，键值对存储。

```
public interface MapState<UK, UV> {
    UV get(UK key) throws Exception;
    void put(UK key, UV value) throws Exception;
```

```
    void putAll(Map<UK, UV> map) throws Exception;
    void remove(UK key) throws Exception;
    boolean contains(UK key) throws Exception;
    Iterable<Map.Entry<UK, UV>> entries() throws Exception;
    Iterable<UK> keys() throws Exception;
    Iterable<UV> values() throws Exception;
    Iterator<Map.Entry<UK, UV>> iterator() throws Exception;
    boolean isEmpty() throws Exception;
}
```

一个实现类：

```
class BatchExecutionKeyMapState<K, N, UK, UV> {
    @Override
    public UV get(UK key) throws Exception {
        if (getCurrentNamespaceValue() == null) {
            return null;
        }
        return getCurrentNamespaceValue().get(key);
    }

    @Override
    public void put(UK key, UV value) {
        initIfNull();
        getCurrentNamespaceValue().put(key, value);
    }

    @Override
    public void putAll(Map<UK, UV> map) {
        // 初始化一个map
        initIfNull();
        this.getCurrentNamespaceValue().putAll(map);
    }

    @Override
    public void remove(UK key) throws Exception {
        if (getCurrentNamespaceValue() == null) {
            return;
        }
        getCurrentNamespaceValue().remove(key);
        if (getCurrentNamespaceValue().isEmpty()) {
            clear();
        }
    }

    @Override
    public boolean contains(UK key) throws Exception {
        return getCurrentNamespaceValue() != null &&
getCurrentNamespaceValue().containsKey(key);
    }

    @Override
```

```
    public Iterable<Map.Entry<UK, UV>> entries() {
        return getCurrentNamespaceValue() == null
                ? Collections.emptySet()
                : getCurrentNamespaceValue().entrySet();
    }

    @Override
    public Iterable<UK> keys() {
        return getCurrentNamespaceValue() == null
                ? Collections.emptySet()
                : getCurrentNamespaceValue().keySet();
    }

    @Override
    public Iterable<UV> values() {
        return getCurrentNamespaceValue() == null
                ? Collections.emptySet()
                : getCurrentNamespaceValue().values();
    }

    @Override
    public Iterator<Map.Entry<UK, UV>> iterator() {
        return getCurrentNamespaceValue() == null
                ? Collections.emptyIterator()
                : getCurrentNamespaceValue().entrySet().iterator();
    }

    @Override
    public boolean isEmpty() {
        return getCurrentNamespaceValue() == null ||
getCurrentNamespaceValue().isEmpty();
    }
}
```

状态分类

状态可以根据是否支持KeyedStream分类：

状态类型	KeyedStream	非KeyedStream
ValueState	√	×
ListState	√	√
ReducingState	√	×
AggregatingState	√	×
MapState	√	×

Keyed状态和算子状态(非Keyed状态)的区别：

- 状态和特定的key是绑定的，在状态中使用map存储key，即每一个key都存在对应的state实例；

- 算子状态和算子的某个特定实例绑定，整个算子只对应一个状态。
- 算子状态需要开发者实现状态的初始化和快照逻辑，即实现CheckpointedFunction接口

状态也可以根据管理状态分类：

- 原始状态：Raw State，即用户自定义的某些属性值，需要手动实现byte数组来进行读写
- 托管状态：Managed State，即flink自带的ValueState等，其正反序列化由Flink框架支持