Flink StateBackend (2) - DefaultOperatorStateBackend

> DefaultOperatorStateBackend 是 StateBackend 中最基础，也是相对比较简单的组件。

简介

DefaultOperatorStateBackend 中维护着 Operator 中不带 Key 的状态，常见的 State 有 ListState / UnionState / BroadcastState，三者的用途各不相同。

State

ListState

ListState 的使用最广泛，比如 Kafka Offsets 就使用了 ListState 来存储每个 TopicPartition 的 offset 状态。在扩容或缩容时，ListState 中维护的元素会重新分配，以 Kafka Offsets 为例：

```
(1) Parallelism = 2
task-1 : List(TopicPartition(partition=1, offset=100), TopicPartition(partition=2, off
set=100))
task-2 : List(TopicPartition(partition=3, offset=100), TopicPartition(partition=4, off
set=100))
```

```
(2) Parallelism = 2 -> 3
task-1 : List(TopicPartition(partition=1, offset=100), TopicPartition(partition=4, off
set=100))
task-2 : List(TopicPartition(partition=2, offset=100))
task-3 : List(TopicPartition(partition=3, offset=100))
```

可以看到，在扩容后，List 中的元素会采取 round-robin 的方式将元素进行重新均匀分配。

UnionState

UnionState 用处较少，我们在生产中使用 Watermark 的时候遇到过相关问题，通过 UnionState 的方式解决了这一问题，详见 FLINK-5601，以上面的 Kafka Offsets 为例，如果作业开启 checkpoint，在重启恢复后，State 分布如下面所示：

```
(1) Parallelism = 2
task-1 : List(TopicPartition(partition=1, offset=100), TopicPartition(partition=2, off
set=100))
```

```
task-2 : List(TopicPartition(partition=3, offset=100), TopicPartition(partition=4, off
set=100))
```

```
(2) Parallelism = 2 -> 3
task-1 : List(TopicPartition(partition=1, offset=100), TopicPartition(partition=2, off
set=100), TopicPartition(partition=3, offset=100), TopicPartition(partition=4, offset=
100))
task-2 : List(TopicPartition(partition=1, offset=100), TopicPartition(partition=2, off
set=100), TopicPartition(partition=3, offset=100), TopicPartition(partition=4, offset=
100))
task-3 : List(TopicPartition(partition=1, offset=100), TopicPartition(partition=2, off
set=100), TopicPartition(partition=3, offset=100), TopicPartition(partition=4, offset=
100))
```

作业的扩容缩容都不会影响 UnionState 的分布，可以看到，重启恢复后每个 Task 中的 UnionState 是之前所有 Task 中 ListState 的集合。

BroadcastState

BroadcastState 一般不会直接使用，而是放在 BroadcastProcessFunction 或 KeyedBroadcastProcessFunction 中使用，请看如下示例：

```
val descriptor = new MapStateDescriptor[Long, String](
      "broadcast-state",
      BasicTypeInfo.LONG_TYPE_INFO.asInstanceOf[TypeInformation[Long]],
      BasicTypeInfo.STRING_TYPE_INFO)

val srcOne = env.addSource(..)
val srcTwo = env.addSource(..)
val broadcast = srcTwo.broadcast(descriptor)
srcOne.connect(broadcast).process(new TestBroadcastProcessFunction)

class TestBroadcastProcessFunction extends KeyedBroadcastProcessFunction[Long, Long, S
tring, String] {

  lazy val localDescriptor = new MapStateDescriptor[Long, String](
    "broadcast-state",
    BasicTypeInfo.LONG_TYPE_INFO.asInstanceOf[TypeInformation[Long]],
    BasicTypeInfo.STRING_TYPE_INFO)

  override def processBroadcastElement(
      value: String,
      ctx: KeyedBroadcastProcessFunction[Long, Long, String, String]#Context,
      out: Collector[String]): Unit = {

    val key = value.split(":")(1).toLong
    ctx.getBroadcastState(localDescriptor).put(key, value)
  }
}
```

srcTwo 直接调用 broadcast 成为 BroadcastStream，然后和 srcOne 进行 connect，之后 srcTwo 发送的每一条数据都会经过 connect 后的所有下游 Task。下游可以通过实现 KeyedBroadcastProcessFunction 来获取 broadcast 的数据并存储为 BroadcastState，具体操作见上面示例。

Operations

Modify

刚刚上面提到，UnionState 和 ListState 的展现形式都是 List，所以在实际存储中，两者也都是使用了 PartitionableListState 来作为 ListState 的实现。 PartitionableListState 的实现非常简单，就是使用 java.util.ArrayList 来封装实际的状态，并提供了一系列的操作方法如 get() / add(String) / update (List) 等。

BroadcastState 内部也是维护了一个 java.util.Map 来封装实际的状态，提供了一系列 Map 的相关操作方法。

Snapshot

由于所有需要 snapshot 都需要实现 SnapshotStrategy 的 snapshot 方法，所以我们直接跳到 DefaultOperatorStateBackendSnapshotStrategy 看相关实现就好。

```
if (!registeredOperatorStates.isEmpty()) {
    for (Map.Entry<String, PartitionableListState<?>> entry : registeredOperatorStates
.entrySet()) {
        PartitionableListState<?> listState = entry.getValue();
        if (null != listState) {
            listState = listState.deepCopy();
        }
        registeredOperatorStatesDeepCopies.put(entry.getKey(), listState);
    }
}

if (!registeredBroadcastStates.isEmpty()) {
    for (Map.Entry<String, BackendWritableBroadcastState<?, ?>> entry : registeredBroa
dcastStates.entrySet()) {
        BackendWritableBroadcastState<?, ?> broadcastState = entry.getValue();
        if (null != broadcastState) {
            broadcastState = broadcastState.deepCopy();
        }
        registeredBroadcastStatesDeepCopies.put(entry.getKey(), broadcastState);
    }
}
```

第一步，先对所有的 PartitionableListState 和 BackendWritableBroadcastState 做 deepCopy，可能你会问，这样效率不是会很低吗？然而事实就是如此，Flink 把 DefaultOperatorStateBackend 中存储的 State 都看作是轻量 State，后续有相对高效的做法，放在 HeapKeyedStateBackend 和 RocksDBKeyedStateBackend 里讲。

这里做 copy 的原因当然是为了后续的异步操作啦

```
AsyncSnapshotCallable<SnapshotResult<OperatorStateHandle>> snapshotCallable =
new AsyncSnapshotCallable<SnapshotResult<OperatorStateHandle>>() {

    @Override
    protected SnapshotResult<OperatorStateHandle> callInternal() throws Exception {

        for (Map.Entry<String, PartitionableListState<?>> entry :
            registeredOperatorStatesDeepCopies.entrySet()) {

            PartitionableListState<?> value = entry.getValue();
            long[] partitionOffsets = value.write(localOut);
            OperatorStateHandle.Mode mode = value.getStateMetaInfo().getAssignmentMode
();
            writtenStatesMetaData.put(
                entry.getKey(),
                new OperatorStateHandle.StateMetaInfo(partitionOffsets, mode));
        }

        for (Map.Entry<String, BackendWritableBroadcastState<?, ?>> entry :
            registeredBroadcastStatesDeepCopies.entrySet()) {

            BackendWritableBroadcastState<?, ?> value = entry.getValue();
            long[] partitionOffsets = {value.write(localOut)};
            OperatorStateHandle.Mode mode = value.getStateMetaInfo().getAssignmentMode
();
            writtenStatesMetaData.put(
                entry.getKey(),
                new OperatorStateHandle.StateMetaInfo(partitionOffsets, mode));
        }
    }
};
```

跳过了写入状态的 metaInfo 的部分，可以看到对于 PartitionableListState 和 BackendWritableBroadcastState 的处理是几乎一样的，以 ListState 为例：

- 写入具体的 ListState 的值，返回 List 中每个元素的 offset
- 用 stateName 和 partitionOffsets 作为 metadata 放入 metadata 集合中

这里存储了 offset 是为了之后恢复时可以快速的定位到需要恢复的状态，写完了对应的状态后，接下来写刚刚存下来的 metadata 集合，整个 Task 的 DefaultOperatorStateBackend 中的状态就算写入完成了，此时会在 Hdfs 上生成一个属于自己的文件。

Recovery

状态的恢复主要逻辑在 JobMaster 的 CheckpointCoordinator 中分配 Task 的状态句柄。Task 使用状态句柄来恢复状态的过程比较简单，在这里就不叙述了，可以看 #OperatorStateRestoreOperation#restore 方法。

我们都知道，当作业完成 checkpoint 之后，在 Hdfs 上是一个个文件(x)，那么当作业扩容或者缩容的时候(p1 -> p2)，如何把以前分配给 p1 并行度的 x 个文件重新分配给 p2 并行度的 Tasks 呢？关键代码可以看 StateAssignmentOperator 中的 reDistributePartitionableStates：

```
OperatorStateRepartitioner opStateRepartitioner = RoundRobinOperatorStateRepartitioner
.INSTANCE;

for (int operatorIndex = 0; operatorIndex < newOperatorIDs.size(); operatorIndex++) {
    OperatorState operatorState = oldOperatorStates.get(operatorIndex);
    int oldParallelism = operatorState.getParallelism();

    OperatorID operatorID = newOperatorIDs.get(operatorIndex);

    newManagedOperatorStates.putAll(applyRepartitioner(
        operatorID,
        opStateRepartitioner,
        oldManagedOperatorStates.get(operatorIndex),
        oldParallelism,
        newParallelism));
}
```

在这里可以看到对于每个 Operator 都使用了 RoundRobinOperatorStateRepartitioner.INSTANCE 来重新分配状态。此时分出了两种情况。

Parallelism 不变

并行度不变，那么 ListState / BroadcastState 的状态都是不受影响的，可以直接从文件中恢复，但是 UnionState 之前提到过，是需要把所有 Task 里的 ListState 都集合到一起，所以在这里只需要对 UnionState 进行重新分配。在分配之前，肯定是先要收集所有的 UnionStates 的 metadata:

```
/**
 * Collect union states from given parallelSubtaskStates.
 */
private Map<String, List<Tuple2<StreamStateHandle, OperatorStateHandle.StateMetaInfo>>
> collectUnionStates(
    List<List<OperatorStateHandle>> parallelSubtaskStates) {

   // stateName -> List(每个并行度subtask的所有 union state)
    Map<String, List<Tuple2<StreamStateHandle, OperatorStateHandle.StateMetaInfo>>> un
ionStates =
        new HashMap<>(parallelSubtaskStates.size());

    // 遍历所有的并行度 subtask state
    for (List<OperatorStateHandle> subTaskState : parallelSubtaskStates) {
        for (OperatorStateHandle operatorStateHandle : subTaskState) {
            if (operatorStateHandle == null) {
                continue;
            }

          // 获取 stateName -> StateMetaInfo 的映射集合
            final Set<Map.Entry<String, OperatorStateHandle.StateMetaInfo>> partitionO
ffsetEntries =
                operatorStateHandle.getStateNameToPartitionOffsets().entrySet();

          // 过滤 UNION 类型的 State
```

```
            partitionOffsetEntries.stream()
                .filter(entry -> entry.getValue().getDistributionMode().equals(Operato
rStateHandle.Mode.UNION))
                .forEach(entry -> {
                   // 把 UNION State 相关信息放入 unionStates 中，每个 stateName 对应的 Lis
t 大小为 并行度 * union state数
                    List<Tuple2<StreamStateHandle, OperatorStateHandle.StateMetaInfo>>
 stateLocations =
                        unionStates.computeIfAbsent(entry.getKey(), k -> new ArrayList
<>(parallelSubtaskStates.size() * partitionOffsetEntries.size()));

                    stateLocations.add(Tuple2.of(operatorStateHandle.getDelegateStateH
andle(), entry.getValue()));
                });
        }
    }

    return unionStates;
}
```

入参的 List<List> parallelSubtaskStates 是所有 subtasks 的 state，大家可以把 List 最里面那层的 List 看成是 OperatorStateHandle 就行了，因为这个集合的 size 一直都是 1。然后开始 repartition union states:

```
/**
 * Repartition UNION state.
 */
private void repartitionUnionState(
        Map<String, List<Tuple2<StreamStateHandle, OperatorStateHandle.StateMetaInfo>>
> unionState,
        List<Map<StreamStateHandle, OperatorStateHandle>> mergeMapList) {

    for (Map<StreamStateHandle, OperatorStateHandle> mergeMap : mergeMapList) {
        for (Map.Entry<String, List<Tuple2<StreamStateHandle, OperatorStateHandle.Stat
eMetaInfo>>> e :
                unionState.entrySet()) {

            for (Tuple2<StreamStateHandle, OperatorStateHandle.StateMetaInfo> handleWi
thMetaInfo : e.getValue()) {
                OperatorStateHandle operatorStateHandle = mergeMap.get(handleWithMetaI
nfo.f0);
                if (operatorStateHandle == null) {
                    operatorStateHandle = new OperatorStreamStateHandle(
                        new HashMap<>(unionState.size()),
                        handleWithMetaInfo.f0);
                    mergeMap.put(handleWithMetaInfo.f0, operatorStateHandle);
                }
                operatorStateHandle.getStateNameToPartitionOffsets().put(e.getKey(), h
andleWithMetaInfo.f1);
            }
        }
    }
}
```

Parallelism 变化

Parallelism 如果发生变化，那么除了 UnionState 之外，ListState 也需要重新分配。刚刚我们在 Snapshot 部分看到，Flink 将 List 每个元素的 offset 都写进了 metadata，所以在重新分配的过程中，可以很容易地知道有总共有多少元素，以及新的 Parallelism 中应该怎么分配。

```
// 通过统计 snapshot 中的元信息算出总共需要分配的元素
int totalPartitions = 0;
for (Tuple2<StreamStateHandle, OperatorStateHandle.StateMetaInfo> offsets : current) {
    totalPartitions += offsets.f1.getOffsets().length;
}

// 每个 Task 需要分配的元素
int baseFraction = totalPartitions / newParallelism;
// 无法整除剩余的元素，采用先到先得的原则分配
int remainder = totalPartitions % newParallelism;
```