layout: title title: Flink checkpoint 机制源码解析 date: 2018/10/16 20:46:25 tags: Flink

前言

Flink checkpoint是Flink Fault Tolerance机制的重要构成部分，该机制确保即使存在故障，程序的每条记录只会作用于状态一次（exactly-once），当然也可以降级为至少一次（at-least-once）。 那么这个机制在源码里是怎么实现的呢？ 在这里我们肯会想到几个问题：

1. Checkpoint请求是由谁发起的?
2. Checkpoint请求是在什么情况下发起的?
3. Checkpoint请求的的接受者是怎么接受到这个请求的?
4. 收到Checkpoint之后做了什么事?
5. Checkpoint请求的成功与失败是如何被发起者感知的？ 带着上面的问题我们来来看看flink是怎么实现的。

Checkpoint 的协调与发起

首先回答第一个问题，checkpoint请求 统一由 JobManager 发起，那么JM是如何发起这个请求的呢？ 我们来看相关逻辑：在这里需要大家先接触到第一个非常关键的类——CheckpointCoordinator。该类可以理解为检查点的协调器，用来协调operator和state的分布式快照。

CheckpointCoordinator

flink 的 checkpoint 统一由 CheckpointCoordinator 来协调，通过将 checkpoint 命令事件发送给相关的 tasks 【源 tasks】，它发起 checkpoint 并且收集 checkpoint 的 ack 消息。

构造参数

这里有必要了解一些关键参数，以便我们更加了解 Checkpoint 的细节策略

- baseInternal：快照的间隔
- checkpointTimeout：一次 checkpoint 的超时时间，超时的 checkpoint 会被取消
- maxConcurrentCheckpointAttempts：最多可同时存在的 checkpoint 任务，是对于整个 flink job
- tasksToTrigger：触发分布式 Checkpoint 的起始 tasks，也就是 source tasks

Checkpoint的发起

前面我们说过checkpoint请求 统一由 JobManager 发起，那么我们来看看JobManager是如何做的？ 我们都知道flink内部是以Actor作为网络通讯框架的，JobManager跟TaskManager都实现了继承了Actor类，实现了其最重要的消息处理方法：

```
/** Handle incoming messages
  *
  * @return
  */
def handleMessage: Receive
```

所有JM跟TM的请求交互都是通过这个方法进行的，自然触发checkpoint的请求也是通过这个方法。JM在收到类型为AbstractCheckpointMessage的消息的时候会判断Checkpoint消息类型的不同进行不同的Checkpoint请求操作。AbstractCheckpointMessage这个是Checkpoint消息的基类，其下有四个实现类，分别是：

- AcknowledgeCheckpoint：代表检查点发起执行成功
- DeclineCheckpoint：代表检查点暂时无法执行
- NotifyCheckpointComplete：代表检查点执行完成
- TriggerCheckpoint：代表触发检查点执行请求

对应的JM收到AbstractCheckpointMessage的消息的处理代码如下，

```
 case checkpointMessage : AbstractCheckpointMessage =>
handleCheckpointMessage(checkpointMessage)
```

我们等下看这个handleCheckpointMessage方法的实现。因为我们这时候要问一个问题：JM什么时候会收到AbstractCheckpointMessage类型的消息？

JobManager 提交作业的时候会根据JobGraph构建ExecutionGraph【flink物理计划抽象】具体见`JobManager line1213 submitJob`。在ExecutionGraph中会构建我们上面提到的CheckpointCoordinator。然后发起检查点的操作！ 首先看看在submitJob中是如何构建ExecutionGraph的，

```
//... JobManager.scala line1277
    executionGraph = ExecutionGraphBuilder.buildGraph(
      executionGraph,
      jobGraph,
      flinkConfiguration,
      futureExecutor,
      ioExecutor,
      scheduler,
      userCodeLoader,
      checkpointRecoveryFactory,
      Time.of(timeout.length, timeout.unit),
      restartStrategy,
      jobMetrics,
      numSlots,
      blobServer,
      log.logger)
//...
```

其调用的是ExecutionGraphBuilder.buildGraph方法，在构造好ExecutionGraph后，又调用了ExecutionGraph的核心的方法enableCheckpointing来开启开启检查点。

```java
//... ExecutionGraphBuilder.java line 291
    executionGraph.enableCheckpointing(
     chkConfig.getCheckpointInterval(),
     chkConfig.getCheckpointTimeout(),
     chkConfig.getMinPauseBetweenCheckpoints(),
     chkConfig.getMaxConcurrentCheckpoints(),
     chkConfig.getExternalizedCheckpointSettings(),
     triggerVertices,
     ackVertices,
     confirmVertices,
     hooks,
     checkpointIdCounter,
     completedCheckpoints,
     externalizedCheckpointsDir,
     metadataBackend,
     checkpointStatsTracker);
//...
```

接下来我们深入到这个方法内部看看它做了什么?

```
//... ExecutionGraph.java line 447
  // create the coordinator that triggers and commits checkpoints
and holds the state 创建触发和提交检查点并保持状态的协调器
    checkpointCoordinator = new CheckpointCoordinator(
        jobInformation.getJobId(),
        interval,
        checkpointTimeout,
        minPauseBetweenCheckpoints,
        maxConcurrentCheckpoints,
        externalizeSettings,
        tasksToTrigger,
        tasksToWaitFor,
        tasksToCommitTo,
        checkpointIDCounter,
        checkpointStore,
        checkpointDir,
        ioExecutor,
        SharedStateRegistry.DEFAULT_FACTORY);
//...

//   注册作业状态检查监听器。所有的监听器均保存在ExecutionGraph成员变量
List<JobStatusListener> jobStatusListeners 里面。
    if (interval != Long.MAX_VALUE) {
        // the periodic checkpoint scheduler is activated and
deactivated as a result of
        // job status changes (running -> on, all other states ->
off)

registerJobStatusListener(checkpointCoordinator.createActivatorDeac
tivator());
    }

public void registerJobStatusListener(JobStatusListener listener) {
    if (listener != null) {
        jobStatusListeners.add(listener);
    }
}
```

上面的方法主要做了两件事初始化**CheckpointCoordinator**和注册作业状态检查监听器；checkpointCoordinator.createActivatorDeactivator()方法返回的就是我们需要注意的一个非常重要的类 CheckpointCoordinatorDeActivator，它负责监听作业的启停，并在作业 启动的时候启动checkpoint请求。请看下面的**CheckpointCoordinatorDeActivator**代码：

```
public class CheckpointCoordinatorDeActivator implements
JobStatusListener {

    private final CheckpointCoordinator coordinator;

    public CheckpointCoordinatorDeActivator(CheckpointCoordinator
coordinator) {
        this.coordinator = checkNotNull(coordinator);
    }

    @Override
    public void jobStatusChanges(JobID jobId, JobStatus
newJobStatus, long timestamp, Throwable error) {
        if (newJobStatus == JobStatus.RUNNING) {
            // start the checkpoint scheduler 开始周期发起检查点
            coordinator.startCheckpointScheduler();
        } else {
            // anything else should stop the trigger for now
            coordinator.stopCheckpointScheduler();
        }
    }
}
```

通过上面的代码我们已经知道要想发起检查点必须等待作业的状态变为RUNNING，CheckpointCoordinatorDeActivator才会感知到从而发起执行检查点。

那么作业的状态什么时候变为RUNNING呢？ CheckpointCoordinatorDeActivator的方法jobStatusChanges在哪里被执行了呢？

注意，别忘了我们目前一直在JobManager的submitJob中。前面的代码可以简单的理解为在一直构建作业的ExecutionGraph。然后就是提交ExecutionGraph。在提交成功后，调用了ExecutionGraph的scheduleForExecution方法；这个方法里 会调用ExecutionGraph持有的所有监听器。更加详细的调用关系请参看下面的代码；

```
//... ExecutionGraph.java line 843
public void scheduleForExecution() throws JobException {
        if (transitionState(JobStatus.CREATED, JobStatus.RUNNING))
{
            switch (scheduleMode) {
                case LAZY_FROM_SOURCES:
                    scheduleLazy(slotProvider);
```

```java
                    break;
                case EAGER:
                    scheduleEager(slotProvider,
scheduleAllocationTimeout);
                    break;
                default:
                    throw new JobException("Schedule mode is
invalid.");
            }
        }
        else {
            throw new IllegalStateException("Job may only be
scheduled from state " + JobStatus.CREATED);
        }
    }

private boolean transitionState(JobStatus current, JobStatus
newState, Throwable error) {
        // consistency check
        if (current.isTerminalState()) {
            String message = "Job is trying to leave terminal state
" + current;
            LOG.error(message);
            throw new IllegalStateException(message);
        }
        // now do the actual state transition
        if (STATE_UPDATER.compareAndSet(this, current, newState)) {
            LOG.info("Job {} ({}) switched from state {} to {}.",
getJobName(), getJobID(), current, newState, error);

            stateTimestamps[newState.ordinal()] =
System.currentTimeMillis();
            notifyJobStatusChange(newState, error);
            return true;
        }
        else {
            return false;
        }
    }

private void notifyJobStatusChange(JobStatus newState, Throwable
error) {
    if (jobStatusListeners.size() > 0) {
        final long timestamp = System.currentTimeMillis();
        final Throwable serializedError = error == null ? null :
new SerializedThrowable(error);
        for (JobStatusListener listener : jobStatusListeners) {
            try {
                listener.jobStatusChanges(getJobID(), newState,
timestamp, serializedError);
            } catch (Throwable t) {
                LOG.warn("Error while notifying JobStatusListener",
t);
            }
        }
    }
}
```

从上面的代码可以看出在ExecutionGraph的方法notifyJobStatusChange里面触发了状态的改变从而发起了checkpoint；那么我们不防将目光聚焦到JobStatusListener(当前是CheckpointCoordinatorDeActivator)的jobStatusChanges的内部，看它到底做了什么？CheckpointCoordinatorDeActivator的成员变量是CheckpointCoordinator，在作业状态变为RUNING的是调用了CheckpointCoordinator的方法startCheckpointScheduler。 接下来让我们从这个方法开始进入到CheckpointCoordinator内部去了解它是怎么协调检查点的。首先先看下这个方法的代码。

```
//... CheckpointCoordinator.java line 843 被
CheckpointCoordinatorDeActivator的方法jobStatusChanges吊起
public void startCheckpointScheduler() {
    synchronized (lock) {
        if (shutdown) {
            throw new IllegalArgumentException("Checkpoint
coordinator is shut down");
        }
        // make sure all prior timers are cancelled 确保所有之前的计时
器都被取消
        stopCheckpointScheduler();

        periodicScheduling = true;
        currentPeriodicTrigger = timer.scheduleAtFixedRate(new
ScheduledTrigger(),  baseInterval, baseInterval,
TimeUnit.MILLISECONDS);
    }
}
```

上面的代码可以看出flink的checkpoint是由CheckpointCoordinator内部的一个timer线程池定时产生的，具体代码由ScheduledTrigger这个Runnable类启动。它会发起一个 timer task，定时执行，并且传入的时间为当前的系统时间，由于 CheckpointCoordinator 全局只有一个，这个时间也是全局递增并且唯一的：

```
//CheckpointCoordinator line 1020
private class ScheduledTrigger extends TimerTask {

        @Override
        public void run() {
            try {
                triggerCheckpoint(System.currentTimeMillis());
            }
            catch (Exception e) {
                LOG.error("Exception while triggering checkpoint",
e);
            }
        }
    }
```

下面我们来具体分析 triggerCheckpoint 这个核心方法

checkpoint 的触发

```
//CheckpointCoordinator line389
public boolean triggerCheckpoint(long timestamp, long
nextCheckpointId) throws Exception {
        // make some eager pre-checks
        synchronized (lock) {
            // abort if the coordinator has been shutdown in the
meantime
            if (shutdown) {
                return false;
            }
             //.......
        // send the messages to the tasks that trigger their
checkpoint 将消息发送到触发其检查点的任务
        for (Execution execution : executions) {
            execution.triggerCheckpoint(checkpointID, timestamp,
checkpointOptions);
        }

//...
```

整个方法代码太长就不占用篇幅了，大家可以结合我的总结去看源码，整个triggerCheckpoint方法大致分为三个部分:

1. 环境前置检查，主要在生成一个chepoint之前进行了一些pre-checks，包括checkpoint的targetDirectory、正在进行中的pendingCheckpoint数量上限、前后两次checkpoint间隔是否过小、以及下游与checkpoint相关tasks是否存活等检测，任意一个条件不满足的都不会执行真正的checkpoint动作。
2. 生成pendingcheckpoint，pendingcheckpoint表示一个待处理的检查点，每个pendingcheckpoint标有一个全局唯一的递增checkpointID，并声明了一个canceller用于后续超时情况下的checkpoint清理用于释放资源。
3. pendingcheckpoint在正式执行前还会再执行一遍前置检查，主要等待完成的检查点数量是否过多以及前后两次完成的检查点间隔是否过短等问题，这些检查都通过后，会把之前定义好的cancller注册到timer线程池，如果等待时间过长会主动回收checkpoint的资源。
4. 启动checkpoint执行，发送这个checkpoint的checkpointID和timestamp到各个对应的source executor,也就是给各个TaskManger发一个TriggerCheckpoint类型的消息。

下面的代码就是各个TaskManger发一个TriggerCheckpoint类型的消息的代码:

```java
/** Execution line 824
 * Trigger a new checkpoint on the task of this execution.
 *
 * @param checkpointId of th checkpoint to trigger
 * @param timestamp of the checkpoint to trigger
 * @param checkpointOptions of the checkpoint to trigger
 */
public void triggerCheckpoint(long checkpointId, long timestamp,
CheckpointOptions checkpointOptions) {
    final SimpleSlot slot = assignedResource;

    if (slot != null) {
        final TaskManagerGateway taskManagerGateway =
slot.getTaskManagerGateway();

        taskManagerGateway.triggerCheckpoint(attemptId,
getVertex().getJobId(), checkpointId, timestamp,
checkpointOptions);
    } else {
        LOG.debug("The execution has no slot assigned. This
indicates that the execution is " +
            "no longer running.");
    }
}
  //...........triggerCheckpoint有很多重载.............
@Override
public void triggerCheckpoint(
        ExecutionAttemptID executionAttemptID,
        JobID jobId,
        long checkpointId,
        long timestamp,
        CheckpointOptions checkpointOptions) {

    Preconditions.checkNotNull(executionAttemptID);
    Preconditions.checkNotNull(jobId);
    actorGateway.tell(new TriggerCheckpoint(jobId,
executionAttemptID, checkpointId, timestamp, checkpointOptions));
}
```

注意源码里其中，for (Execution execution: executions) 这里面的executions里面是所有的输入节点，也就是flink source节点，所以checkpoint这些barrier 时间首先从jobmanager发送给了所有的source task！ 总结逻辑：

- 如果已关闭或优先处理排队请求会总额并发任务超过限制，都会取消此次 checkpoint 的发起
- 如果最小间隔时间未达到，也会取消此次 checkpoint
- check 所有的发起节点【源节点】与其他节点都为 RUNNING 状态后才会发起 checkpoint
- 发起 checkpoint 并生成一个 PendingCheckpoint 【已经发起但尚未 ack 的 checkpoint】
- 每个源节点都会发一条消息给自己的 TaskManager 进行 checkpoint

Checkpoint 消息的处理

前面已经说明JM端是如何发起checkpoint的，这节我们来看看TM端是怎么收到TriggerCheckpoint消息并处理的. TriggerCheckpoint消息进入TaskManager的处理路径为

```
handleMessage -> handleCheckpointingMessage ->
Task.triggerCheckpointBarrier.
```

具体见 `TaskManager.scala line274 handleMessage` 。 我们来看看TM端收到TriggerCheckpoint消息的核心方法：

```
//TaskManager.scala line 496
private def handleCheckpointingMessage(actorMessage:
AbstractCheckpointMessage): Unit = {
    actorMessage match {
      case message: TriggerCheckpoint =>
        val taskExecutionId = message.getTaskExecutionId
        val checkpointId = message.getCheckpointId
        val timestamp = message.getTimestamp

        log.debug(s"Receiver TriggerCheckpoint
$checkpointId@$timestamp for $taskExecutionId.")
        val task = runningTasks.get(taskExecutionId)
        if (task != null) {
          task.triggerCheckpointBarrier(checkpointId, timestamp,
checkpointOptions)
        } else {
          log.debug(s"TaskManager received a checkpoint request for
unknown task $taskExecutionId.")
        }
//...
```

```
//Task.java line 1176
public void triggerCheckpointBarrier(final long checkpointID, final
long checkpointTimestamp) {
        AbstractInvokable invokable = this.invokable;
        if (executionState == ExecutionState.RUNNING && invokable
!= null) {
            if (invokable instanceof StatefulTask) {
                // build a local closure
                final StatefulTask<?> statefulTask =
(StatefulTask<?>) invokable;
                final String taskName = taskNameWithSubtask;
                boolean success =
statefulTask.triggerCheckpoint(checkpointMetaData,
checkpointOptions);
//...
//StreamTask line577
protected boolean performCheckpoint(final long checkpointId, final
long timestamp) throws Exception {
        LOG.debug("Starting checkpoint {} on task {}",
checkpointId, getName());
        synchronized (lock) {
            if (isRunning) {
                operatorChain.broadcastCheckpointBarrier(

checkpointMetaData.getCheckpointId(),

checkpointMetaData.getTimestamp(),
                                        checkpointOptions);
            checkpointState(checkpointMetaData, checkpointOptions,
checkpointMetrics);
            return true;
            }
```

在正常的情况下，triggerCheckpointBarrier会调用StreamTask内部实现的triggerCheckpoint()方法，并根据调用链条

```
triggerCheckpoint->performCheckpoint->checkpointState-
>CheckpointingOperation.executeCheckpointing
```

调用checkpointState方法之前，flink会把预先把checkpointBarrier发送到下游task，以便下游operator尽快开始他们的checkpoint进程， 这个时候大家有没有注意到Checkpoint的执行很快进入到一个非常重要的类：StreamTask里面。这个类是所有流处理任务的基类，实现位于flink-streaming-java模块中；关于这个类在flink状态管理会详细介绍。这个时候真正进入到task级别的Checkpoint的执行。 下面我们深入到StreamTask内部CheckpointingOperation类的方法executeCheckpointing方法内部看看其是如何执行的？CheckpointingOperation封装了operate的检查点点的执行逻辑。

```
//StreamTask line 1033

 public void executeCheckpointing() throws Exception {
        startSyncPartNano = System.nanoTime();
        boolean failed = true;
        try {
            for (StreamOperator<?> op : allOperators) {
                checkpointStreamOperator(op);   //调用
其snapshotState方法
            }
            if (LOG.isDebugEnabled()) {
                LOG.debug("Finished synchronous checkpoints for
checkpoint {} on task {}",
                        checkpointMetaData.getCheckpointId(),
owner.getName());
            }
            startAsyncPartNano = System.nanoTime();

checkpointMetrics.setSyncDurationMillis((startAsyncPartNano -
startSyncPartNano) / 1_000_000);
            // at this point we are transferring ownership over
snapshotInProgressList for cleanup to the thread
            runAsyncCheckpointingAndAcknowledge(); //通知JM的ck发起成
功，JM接收AcknowledgeCheckpoint消息并检查=此消息是否与挂起检查点相关联。调用
的是CheckpointCoordinator的receiveAcknowledgeMessage方法~
            failed = false;
            if (LOG.isDebugEnabled()) {
                LOG.debug("{} - finished synchronous part of
checkpoint {}." +
                        "Alignment duration: {} ms, snapshot
duration {} ms",
                    owner.getName(),
checkpointMetaData.getCheckpointId(),
                    checkpointMetrics.getAlignmentDurationNanos() /
1_000_000,
                    checkpointMetrics.getSyncDurationMillis());
                }
                //.......
```

在executeCheckpointing方法里进行了两个操作，

1. 是对该task对应的所有StreamOperator对象调用checkpointStreamOperator(op)，也就是每个operate执行自己的checkpoint操作。
2. 是通知JobManager Checkpoint 发起成功了~

下面看下checkpointStreamOperator这个方法的实现：

```
 private void checkpointStreamOperator(StreamOperator<?> op) throws
Exception {
      if (null != op) {
        OperatorSnapshotResult snapshotInProgress =
op.snapshotState(
                checkpointMetaData.getCheckpointId(),
                checkpointMetaData.getTimestamp(),
                checkpointOptions);
        operatorSnapshotsInProgress.put(op.getOperatorID(),
snapshotInProgress);
    }
```

这个方法很简单了就是调用具体的operate的snapshotState方法，然后使用operatorSnapshotsInProgress这个集合收集每个op的执行进度的。
StreamOperator的snapshotState(long checkpointId,long timestamp,CheckpointOptions checkpointOptions)方法最终由它的子类AbstractStreamOperator给出了一个final实现

```
 @Override
public final OperatorSnapshotResult snapshotState(long
checkpointId, long timestamp, CheckpointOptions checkpointOptions)
throws Exception {
        KeyGroupRange keyGroupRange = null != keyedStateBackend ?
                keyedStateBackend.getKeyGroupRange() :
KeyGroupRange.EMPTY_KEY_GROUP_RANGE;

        OperatorSnapshotResult snapshotInProgress = new
OperatorSnapshotResult();

        CheckpointStreamFactory factory =
getCheckpointStreamFactory(checkpointOptions);

        try (StateSnapshotContextSynchronousImpl snapshotContext =
new StateSnapshotContextSynchronousImpl(
                checkpointId,
                timestamp,
                factory,
                keyGroupRange,
                getContainingTask().getCancelables())) {

            snapshotState(snapshotContext);
snapshotInProgress.setKeyedStateRawFuture(snapshotContext.getKeyedS
tateStreamFuture());

snapshotInProgress.setOperatorStateRawFuture(snapshotContext.getOpe
ratorStateStreamFuture());

            if (null != operatorStateBackend) {
                snapshotInProgress.setOperatorStateManagedFuture(
                    operatorStateBackend.snapshot(checkpointId,
timestamp, factory, checkpointOptions));
            }

            if (null != keyedStateBackend) {
                snapshotInProgress.setKeyedStateManagedFuture(
                    keyedStateBackend.snapshot(checkpointId,
timestamp, factory, checkpointOptions));
            }
        } catch (Exception snapshotException) {
            try {
                snapshotInProgress.cancel();
            } catch (Exception e) {
                snapshotException.addSuppressed(e);
            }

            throw new Exception("Could not complete snapshot " +
checkpointId + " for operator " +
                getOperatorName() + '.', snapshotException);
        }

        return snapshotInProgress;
    }
```

上述代码里的snapshotState(snapshotContext)方法在不同的最终operator中有自己的具体实现。 executeCheckpointing的第二个操作是然后是调用runAsyncCheckpointingAndAcknowledge执行所有的state固化文件操作并返回acknowledgeCheckpoint给JobManager。

应答 Checkpoint 消息

下面我们来看看task是如何应答checkpoint的。其主要方法是runAsyncCheckpointingAndAcknowledge();前面我们讲过checkpoint的发起是调用的StreamingTask的triggerCheckpoint方法其内部调用performCheckpoint方法。performCheckpoint内部首先先将此次 checkpoint 的 barrier 广播到下游，以便让下游快速 checkpoint;

```
//StreamTask line 586
private boolean performCheckpoint(
        CheckpointMetaData checkpointMetaData,
        CheckpointOptions checkpointOptions,
        CheckpointMetrics checkpointMetrics) throws Exception {
//....... checkpoint 的 barrier 广播到下游，以便让下游快速 checkpoint;
        operatorChain.broadcastCheckpointBarrier(
                        checkpointMetaData.getCheckpointId(),
                        checkpointMetaData.getTimestamp(),
                        checkpointOptions);

        checkpointState(checkpointMetaData, checkpointOptions,
checkpointMetrics);
        return true;
//.......
}
```

这里一个重要的方法checkpointState，这个方法主要做了两件事，一是让每个operater保存自己的状态，二是向JobManager响应这个checkpoint。下面我们分别看看他们是怎么实现的? 下面给出checkpointState方法的调用链：

```
performCheckpoint-->checkpointState--
>checkpointingOperation.executeCheckpointing()--
>checkpointStreamOperator(op)-->
runAsyncCheckpointingAndAcknowledge();
```

checkpointStreamOperator(op)前面已经讲过了，我们目前主要看runAsyncCheckpointingAndAcknowledge这个方法是如何应答JobManager的。下面给出这个方法的代码；

```
//StreamTask line 586
public void runAsyncCheckpointingAndAcknowledge() throws
IOException {
    AsyncCheckpointRunnable asyncCheckpointRunnable = new
AsyncCheckpointRunnable(
            owner,
            operatorSnapshotsInProgress,
            checkpointMetaData,
            checkpointMetrics,
            startAsyncPartNano);

    owner.cancelables.registerCloseable(asyncCheckpointRunnable);

owner.asyncOperationsThreadPool.submit(asyncCheckpointRunnable);
}
```

这个方法最终还是委托**AsyncCheckpointRunnable**，那么我们来看看这个类内部做了什么？

```
//StreamTask line 850
private static final class AsyncCheckpointRunnable implements
Runnable, Closeable {
//TaskStateSnapshot:非常重要，她封装了每个operate对应的state的句柄，将来state恢复的时候使用的也是这么对象，记住这个对象是在这里创建的！他会被用来构造AcknowledgeCheckpoint这个消息，然后JM传给CheckpointCoordinator
    final TaskStateSnapshot taskOperatorSubtaskStates = new
TaskStateSnapshot(operatorSnapshotsInProgress.size());
   //...... 第一步获取每个operater的checkpoint的执行结果
   for (Map.Entry<OperatorID, OperatorSnapshotResult> entry :
operatorSnapshotsInProgress.entrySet()) {
            OperatorID operatorID = entry.getKey();
            OperatorSnapshotResult snapshotInProgress =
entry.getValue();

            OperatorSubtaskState operatorSubtaskState = new
OperatorSubtaskState(

FutureUtil.runIfNotDoneAndGet(snapshotInProgress.getOperatorStateMa
nagedFuture()),

FutureUtil.runIfNotDoneAndGet(snapshotInProgress.getOperatorStateRa
wFuture()),

FutureUtil.runIfNotDoneAndGet(snapshotInProgress.getKeyedStateManag
edFuture()),
FutureUtil.runIfNotDoneAndGet(snapshotInProgress.getKeyedStateRawFu

ture())
            );

            hasState |= operatorSubtaskState.hasState();

taskOperatorSubtaskStates.putSubtaskStateByOperatorID(operatorID,
operatorSubtaskState); //使用TaskStateSnapshot封装operator的state的执
行句柄
        }
   //......  第二步 发现operater的checkpoint的状态为COMPLETED，那么就应答
JobManager
    if
(asyncCheckpointState.compareAndSet(CheckpointingOperation.AsynChec
kpointState.RUNNING,
            CheckpointingOperation.AsynCheckpointState.COMPLETED))
{

        TaskStateSnapshot acknowledgedState = hasState ?
taskOperatorSubtaskStates : null;

        // we signal stateless tasks by reporting null, so that
there are no attempts to assign empty state
        // to stateless tasks on restore. This enables simple job
modifications that only concern
        // stateless without the need to assign them uids to match
their (always empty) states.
        owner.getEnvironment().acknowledgeCheckpoint(
            checkpointMetaData.getCheckpointId(),
            checkpointMetrics,
            acknowledgedState);

        LOG.debug("{} - finished asynchronous part of checkpoint
{}. Asynchronous duration: {} ms",
            owner.getName(), checkpointMetaData.getCheckpointId(),
asyncDurationMillis);

        LOG.trace("{} - reported the following states in snapshot
for checkpoint {}: {}.",
            owner.getName(), checkpointMetaData.getCheckpointId(),
acknowledgedState);

    } else {
        LOG.debug("{} - asynchronous part of checkpoint {} could
not be completed because it was closed before.",
            owner.getName(),
            checkpointMetaData.getCheckpointId());
    }
}
```

上面的AsyncCheckpointRunnable做要做了两个事，第一步获取每个operater的checkpoint的执行结果(这些信息事在方法checkpointStreamOperator(op)放进去的)，使用TaskStateSnapshot封装每个operator的state的执行句柄 第二步 发现operater的checkpoint的状态为COMPLETED，那么就应答JobManager，我们来重点关注这个代码的内部执行逻辑。

```
// we signal stateless tasks by reporting null, so that there are
no attempts to assign empty state
// to stateless tasks on restore. This enables simple job
modifications that only concern
// stateless without the need to assign them uids to match their
(always empty) states.
owner.getEnvironment().acknowledgeCheckpoint(
    checkpointMetaData.getCheckpointId(),
    checkpointMetrics,
    acknowledgedState);
```

owner是AsyncCheckpointRunnable的属性StreamTask.owner.getEnvironment()是RuntimeEnvironment类的acknowledgeCheckpoint方法。这个方法其实会封装一个AcknowledgeCheckpoint类型的消息(这个消息包含TaskStateSnapshot)返回给JobManager。

```
//RuntimeEnvironment line 245
@Override
public void acknowledgeCheckpoint(
        long checkpointId,
        CheckpointMetrics checkpointMetrics,
        TaskStateSnapshot checkpointStateHandles) {

    checkpointResponder.acknowledgeCheckpoint(
            jobId, executionId, checkpointId, checkpointMetrics,
            checkpointStateHandles);
}
```

RuntimeEnvironment类持有的对象ActorGatewayCheckpointResponder(上文的checkpointResponder)用来跟JobManager通讯。

```
//ActorGatewayCheckpointResponder line42
public class ActorGatewayCheckpointResponder implements
CheckpointResponder {

    private final ActorGateway actorGateway;

    public ActorGatewayCheckpointResponder(ActorGateway
actorGateway) {
        this.actorGateway =
Preconditions.checkNotNull(actorGateway);
    }

    @Override
    public void acknowledgeCheckpoint(
            JobID jobID,
            ExecutionAttemptID executionAttemptID,
            long checkpointId,
            CheckpointMetrics checkpointMetrics,
            TaskStateSnapshot checkpointStateHandles) {

        AcknowledgeCheckpoint message = new AcknowledgeCheckpoint(
                jobID, executionAttemptID, checkpointId,
checkpointMetrics,
                checkpointStateHandles);

        actorGateway.tell(message);
    }
```

这里需要介绍下AcknowledgeCheckpoint这个消息，该消息是一个应答信号，表示某个独立的task的检查点已经完成。也是由TaskManager发送给JobManager。该消息会携带task的状态：state，stateSize； 那么JobManager是怎么收到这个消息的呢？收到以后又是怎么处理的呢？让我们依然回到JobManager的方法handleCheckpointMessage来找答案；

```
//JobManager line1415
  private def handleCheckpointMessage(actorMessage:
AbstractCheckpointMessage): Unit = {
    actorMessage match {
      case ackMessage: AcknowledgeCheckpoint =>
        val jid = ackMessage.getJob()
        currentJobs.get(jid) match {
          case Some((graph, _)) =>
            val checkpointCoordinator =
graph.getCheckpointCoordinator()

            if (checkpointCoordinator != null) {
              future {
                try {
                  if
(!checkpointCoordinator.receiveAcknowledgeMessage(ackMessage)) {
                    log.info("Received message for non-existing
checkpoint " +
                      ackMessage.getCheckpointId)
                  }
                }
                catch {
                  case t: Throwable =>
                    log.error(s"Error in CheckpointCoordinator
while processing $ackMessage", t)
                }
              }(context.dispatcher)
            }
            else {
              log.error(
                s"Received AcknowledgeCheckpoint message for job
$jid with no " +
                  s"CheckpointCoordinator")
            }

          case None => log.error(s"Received AcknowledgeCheckpoint
for unavailable job $jid")
        }
```

handleCheckpointMessage方法原来处理TM返回的各种msg,当它发现消息类型是AcknowledgeCheckpoint时候，会找到当前job对应的checkpointCoordinator然后调用其receiveAcknowledgeMessage方法。饶了一圈又回到了checkpointCoordinator这个类。 那么我们就来看看checkpointCoordinator的这个receiveAcknowledgeMessage方法干了什么吧？ 首先判断当前接收到的消息中包含的检查点是否是待处理的检查点。如果是，并且也没有discard掉，则执行如下逻辑：

```
//checkpointCoordinator line 870
if (checkpoint != null && !checkpoint.isDiscarded()) {
    switch
(checkpoint.acknowledgeTask(message.getTaskExecutionId(),
message.getSubtaskState(),
            message.getCheckpointMetrics())) {
        case SUCCESS:
            LOG.debug(
                "Received acknowledge message for checkpoint {}
from task {} of job {}.",
                checkpointId, message.getTaskExecutionId(),
message.getJob());

            if (checkpoint.isFullyAcknowledged()) {
                completePendingCheckpoint(checkpoint);
            }
            break;
//.......
```

检查点首先应答相关的task，这里叫应答不是太贴切，此方法1.删除notYetAcknowledgedTasks记录的task，2.计算这个PendingCheckpoint的包含的所有的operater的state的大小，有operatorStates存储；3.对发布这个CK的metrics信息。 如果检查点已经完全应答完成，则将检查点转换成CompletedCheckpoint，然后将其加入completedCheckpointStore列表，并从pendingCheckpoints中移除。completedCheckpointStore用于后续的restore； 相关逻辑在completePendingCheckpoint这个方法中：

```
// externalize the checkpoint if required CheckpointCoordinator
line946
if (pendingCheckpoint.getProps().externalizeCheckpoint()) {
    completedCheckpoint =
pendingCheckpoint.finalizeCheckpointExternalized();
} else {
    completedCheckpoint =
pendingCheckpoint.finalizeCheckpointNonExternalized();
//.......
// drop those pending checkpoints that are at prior to the
completed one 删除那些在完成的检查点之前的检查点
    dropSubsumedCheckpoints(checkpointId);
}
} finally {
pendingCheckpoints.remove(checkpointId);

triggerQueuedRequests();
}

rememberRecentCheckpointId(checkpointId);

// send the "notify complete" call to all vertices
final long timestamp = completedCheckpoint.getTimestamp();

for (ExecutionVertex ev : tasksToCommitTo) {
    Execution ee = ev.getCurrentExecutionAttempt();
    if (ee != null) {
        ee.notifyCheckpointComplete(checkpointId, timestamp);
    }
}
```

最后通知每一个Execution所在的TaskManager checkpoint做完了。消息类型为NotifyCheckpointComplete

```
/**
 * Notify the task of this execution about a completed checkpoint.
 *
 * @param checkpointId of the completed checkpoint
 * @param timestamp of the completed checkpoint
 */
public void notifyCheckpointComplete(long checkpointId, long
timestamp) {
    final SimpleSlot slot = assignedResource;

    if (slot != null) {
        final TaskManagerGateway taskManagerGateway =
slot.getTaskManagerGateway();

        taskManagerGateway.notifyCheckpointComplete(attemptId,
getVertex().getJobId(), checkpointId, timestamp);
    } else {
        LOG.debug("The execution has no slot assigned. This
indicates that the execution is " +
            "no longer running.");
    }
}
```

那么问题来了TaskManager收到消息类型为NotifyCheckpointComplete的消息时，它做了什么？我们接下来去看TM是怎么做的。我们前面讲过TM的handleCheckpointingMessage负责处理JM发送过来的checkpoint的消息的处理，我们再来复习这个方法。

```
/**  TaskManager line496
   * Handler for messages related to checkpoints.
   *
   * @param actorMessage The checkpoint message.
   */
  private def handleCheckpointingMessage(actorMessage:
AbstractCheckpointMessage): Unit = {

    actorMessage match {
      case message: TriggerCheckpoint =>
        val taskExecutionId = message.getTaskExecutionId
        val checkpointId = message.getCheckpointId
        val timestamp = message.getTimestamp
        val checkpointOptions = message.getCheckpointOptions

        log.debug(s"Receiver TriggerCheckpoint
$checkpointId@$timestamp for $taskExecutionId.")

        val task = runningTasks.get(taskExecutionId)
        if (task != null) {
          task.triggerCheckpointBarrier(checkpointId, timestamp,
checkpointOptions)
        } else {
          log.debug(s"TaskManager received a checkpoint request for
unknown task $taskExecutionId.")
        }

      case message: NotifyCheckpointComplete =>
        val taskExecutionId = message.getTaskExecutionId
        val checkpointId = message.getCheckpointId
        val timestamp = message.getTimestamp

        log.debug(s"Receiver ConfirmCheckpoint
$checkpointId@$timestamp for $taskExecutionId.")

        val task = runningTasks.get(taskExecutionId)
        if (task != null) {
          task.notifyCheckpointComplete(checkpointId)
        } else {
          log.debug(
            s"TaskManager received a checkpoint confirmation for
unknown task $taskExecutionId.")
        }

      // unknown checkpoint message
      case _ => unhandled(actorMessage)
    }
  }
```

上面的方法中TriggerCheckpoint的消息类型最终调用的是StreamTask的triggerCheckpoint方法进行checkpoint的处理。那么消息类型为NotifyCheckpointComplete的相信也是StreamTask的一个方法。‘ 我们顺着代码task.notifyCheckpointComplete(checkpointId)看进去发现调用的正是StreamTask的notifyCheckpointComplete(checkpointID)方法。 既然如此那就好办了，我们来看看StreamTask对于checkpoint做完了是怎么处理的?

```
// StreamTask line643
@Override
public void notifyCheckpointComplete(long checkpointId) throws
Exception {
    synchronized (lock) {
        if (isRunning) {
            LOG.debug("Notification of complete checkpoint for task
{}", getName());

            for (StreamOperator<?> operator :
operatorChain.getAllOperators()) {
                if (operator != null) {

operator.notifyOfCompletedCheckpoint(checkpointId);
                }
            }
        }
        else {
            LOG.debug("Ignoring notification of complete checkpoint
for not-running task {}", getName());
        }
    }
}
```

从上面的代码可以看出 其最终调用的还是这个operator.notifyOfCompletedCheckpoint(checkpointId)方法。这个方法最后被statebackand调用(其中之一),

```
//AbstractStreamOperator line446
@Override
public void notifyOfCompletedCheckpoint(long checkpointId) throws
Exception {
    if (keyedStateBackend != null) {
        keyedStateBackend.notifyCheckpointComplete(checkpointId);
    }
}
```

最后总结其逻辑:

- 先是通过 TaskManager 进行消息路由，对于 TriggerCheckpoint 消息，会路由给相应的 Task 做处理
- Task 会起一个异步 task 进行 checkpoint，内部是调用 StreamTask 的 performCheckpoint 方法
- performCheckpoint 内部首先先将此次 checkpoint 的 barrier 广播到下游，以便让下游快速 checkpoint
- 后执行具体的 checkpoint，将状态持久化，目前支持的持久化方式有：FileSystem、Memory、RocksDB，成功后通知 JobManager 进行 ack，否则取消此次 checkpoint
- 如果是 ack 消息，依据具体情况通知对应的 KVState

最后附一张图描述交互过程：

State的制作

前文说过下面给出checkpointState方法的调用链。
(StreamTask.java)performCheckpoint-->checkpointState-->checkpointingOperation.executeCheckpointing()-->checkpointStreamOperator(op)--> runAsyncCheckpointingAndAcknowledge();
使用的是checkpointStreamOperator(op)这个方法调用operate的snapshotState方法。开始了真正的snapshotState的制作

```
//StreamTask line1089
@SuppressWarnings("deprecation")
private void checkpointStreamOperator(StreamOperator<?> op) throws
Exception {
    if (null != op) {
        OperatorSnapshotResult snapshotInProgress =
op.snapshotState(
        dasdasd        checkpointMetaData.getCheckpointId(),
                checkpointMetaData.getTimestamp(),
                checkpointOptions);
        operatorSnapshotsInProgress.put(op.getOperatorID(),
snapshotInProgress);
    }
}
```

这里需要介绍下StreamOperator，他是所有operater需要实现的接口。处理声明的基本的open、close外，就有我们需要关注的snapshotState方法，我们看下这个方法的声明：

```
//SdtdasreamOperator line101
/**
 * Called to draw a state snapshot from the operator.
 * @return a runnable future to the state handle that points to the
snapshotted state. For synchronous implementations,
 * the runnable might already be finished.
 *
 * @throws Exception exception that happened during snapshotting.
 */
OperatorSnapshotResult snapshotState(
    long checkpointId,
    long timestamp,as
    CheckpointOptions checkpointOptions) throws Exception;
```

OperatorSnapshotResult这个类封装了state的执行的各种Future;也就是由他代表state的执行结果。我们看下它的构造方法：

```
//  OperatorSnapshotResult line101
public OperatorSnapshotResult(
        RunnableFuture<KeyedStateHandle> keyedStateManagedFuture,
        RunnableFuture<KeyedStateHandle> keyedStateRawFuture,
        RunnableFuture<OperatorStateHandle>
operatorStateManagedFuture,
        RunnableFuture<OperatorStateHandle> operatorStateRawFuture)
{
    this.keyedStateManagedFuture = keyedStateManagedFuture;
    this.keyedStateRawFuture = keyedStateRawFuture;
    this.operatorStateManagedFuture = operatorStateManagedFuture;
    this.operatorStateRawFuture = operatorStateRawFuture;
}
```

它包含基本原始的、flink托管的、operate级别的、kv级别的状态Future。这四种是一个operater可能要做的状态。为什么说肯能呢，请看下面： 既然StreamOperator是个接口，那么我们看看它的实现类AbstractStreamOperator对这个方法的实现，AbstractStreamOperator是streamMap、streamFlatMap等的基类。

```
//AbstractStreamOperator line340
@Override
```

```
public final OperatorSnapshotResult snapshotState(long
checkpointId, long timestamp, CheckpointOptions checkpointOptions)

throws Exception {
    KeyGroupRange keyGroupRange = null != keyedStateBackend ?
            keyedStateBackend.getKeyGroupRange() :
KeyGroupRange.EMPTY_KEY_GROUP_RANGE;
    OperatorSnapshotResult snapshotInProgress = new
OperatorSnapshotResult();
    CheckpointStreamFactory factory =
getCheckpointStreamFactory(checkpointOptions);
    try (StateSnapshotContextSynchronousImpl snapshotContext = new
StateSnapshotContextSynchronousImpl(
            checkpointId,
            timestamp,
            factory,
            keyGroupRange,
            getContainingTask().getCancelables())) {
        snapshotState(snapshotContext);

snapshotInProgress.setKeyedStateRawFuture(snapshotContext.getKeyedS
tateStreamFuture());

snapshotInProgress.setOperatorStateRawFuture(snapshotContext.getOpe
ratorStateStreamFuture());
        if (null != operatorStateBackend) {
            snapshotInProgress.setOperatorStateManagedFuture(
                operatorStateBackend.snapshot(checkpointId,
timestamp, factory, checkpointOptions));
        }
        if (null != keyedStateBackend) {
            snapshotInProgress.setKeyedStateManagedFuture(
                keyedStateBackend.snapshot(checkpointId, timestamp,
factory, checkpointOptions));
        }
    } catch (Exception snapshotException) {
        try {
            snapshotInProgress.cancel();
        } catch (Exception e) {
            snapshotException.addSuppressed(e);
        }
        throw new Exception("Could not complete snapshot " +
checkpointId + " for operator " +
            getOperatorName() + '.', snapshotException);
    }
    return snapshotInProgress;
}
```

方法看到这里其实已经很明白了，真正做state的操作已经落到各种StateBackend的身上了。(注意：StateBackend的初始化是在streamTask的invoke方法里初始化的！包括operatorStateBackend、keyedStateBackend) 一个 StateBackend 定义了流程序状态的存储和快照的形式，被 StateBackend 管理的状态会定期进行分布式快照，典型的基础实现[仅仅存储的方式]有： MemoryStateBackend：将工作状态保存在 TaskManager 内存中，并快照到 JobManager 的内存 FsStateBackend：将工作状态保存在 TaskManager 内存中，并快照到文件系统[HDFS/Ceph/S3/GCS] RocksDBStateBackend：将工作状态保存在 RocksDB 中，并快照到文件系统[类似于 FsStateBackend] 由于我们在生产环境中主要使用的是RocksDBStateBackend，那么我们的主要精力还是放在它上面看看它是怎么存储state的。关于RocksDB的跟多的信息请看考：https://www.jianshu.com/p/2638e2b379c3,https://www.jianshu.com/p/061927761027 首先它实现了Snapshotable接口，一个可以执行其状态快照的操作员的接口。所以的backend都实现了其snapshot和restore方法,这两个方法才是真正做Checkpoint和读取Checkpoint恢复的方法。

```java
public interface Snapshotable<S extends StateObject> {
    /**
     * Operation that writes a snapshot into a stream that is provided by the given {@link CheckpointStreamFactory} and
     * returns a @{@link RunnableFuture} that gives a state handle to the snapshot. It is up to the implementation if
     * the operation is performed synchronous or asynchronous. In the later case, the returned Runnable must be executed
     * first before obtaining the handle.
     *
     * @param checkpointId  The ID of the checkpoint.
     * @param timestamp     The timestamp of the checkpoint.
     * @param streamFactory The factory that we can use for writing our state to streams.
     * @param checkpointOptions Options for how to perform this checkpoint.
     * @return A runnable future that will yield a {@link StateObject}.
     */
    RunnableFuture<S> snapshot(
            long checkpointId,
            long timestamp,
            CheckpointStreamFactory streamFactory,
            CheckpointOptions checkpointOptions) throws Exception;

    /**
     * Restores state that was previously snapshotted from the provided parameters. Typically the parameters are state
     * handles from which the old state is read.
     *
     * @param state the old state to restore.
     */
    void restore(Collection<S> state) throws Exception;
}
```

RocksDBStateBackend

RocksDBStateBackend实现了使用文件系统URL(类型，地址，路径)，例如“hdfs://qihoo/flink/checkpoints” 或 “file:///data/flink/checkpoints”. RocksDBStateBackend将数据存储在RocksDB数据库中，它(默认)存储在TaskManager的data目录下。当checkpoint时，整个RocksDB数据库将被checkpoint到配置的文件系统和目录下。最小的元数据(meta)存储在JobManager的内存中(或者，在高可用模式下，在元数据checkpoint中)。 那我们其中最关注的是其snapshot方法。说道这里我们先解决一个问题，那么就是我们在api里使用拿到例如ValueStateDescriptor后的数据是怎么传到后端的RocksDBStateBackend的？ 为了获得一个State句柄，你需要创建一个StateDescriptor，这个StateDescriptor保存了state的名称，State保存的值的类型以及用户自定义函数如:一个ReduceFunction。根据你想要检索的state的类型，你可以创建一个ValueStateDescriptor, 一个ListStateDescriptor, 一个ReducingStateDescriptor, 一个FoldingStateDescriptor或者一个MapStateDescriptor State可以通过RuntimeContext来访问，所以只能在富函数中使用。RichFunction中的RuntimeContext有以下这些方法来访问state:

```
ValueState<T> getState(ValueStateDescriptor<T>)
ReducingState<T> getReducingState(ReduceingStateDescriptor<T>)
ListState<T> getListState(ListStateDescriptor<T>)
FoldingState<T, ACC> getFoldingState(FoldingStateDescriptor<T,
ACC>)
MapState<UK, UV> getMapState(MapStateDescriptor<UK, UV>)
```

RuntimeContext同样是个接口，她定义是了上面关于state的方法，她的方法是怎么跟StateBackend关联的呢？我们来看她的实现类：StreamingRuntimeContext，例如她重写了上面的方法。StreamingRuntimeContext是在operater的setUp的方法里初始化的。 我们看下他是怎么实现getListState?

```
//  StreamingRuntimeContext line124
@Override
public <T> ListState<T> getListState(ListStateDescriptor<T>
stateProperties) {
    KeyedStateStore keyedStateStore =
checkPreconditionsAndGetKeyedStateStore(stateProperties);

stateProperties.initializeSerializerUnlessSet(getExecutionConfig())
;
    return keyedStateStore.getListState(stateProperties);
}
```

大家肯定注意到了一个类KeyedStateStore，他是一个提供注册和管理state的接口。也是提供个获取各种state的方法。其默认实现是：DefaultKeyedStateStore.java.在DefaultKeyedStateStore的构造方法里面传入了StateBackend。所有真正实现state的任务在这里落到了DefaultKeyedStateStore身上。我们来看下她的构造方法：

```java
//  DefaultKeyedStateStore line50
public DefaultKeyedStateStore(KeyedStateBackend<?>
keyedStateBackend, ExecutionConfig executionConfig) {
    this.keyedStateBackend =
Preconditions.checkNotNull(keyedStateBackend);
    this.executionConfig =
Preconditions.checkNotNull(executionConfig);
}

@Override
public <T> ValueState<T> getState(ValueStateDescriptor<T>
stateProperties) {
    requireNonNull(stateProperties, "The state properties must not
be null");
    try {

stateProperties.initializeSerializerUnlessSet(executionConfig);
        return getPartitionedState(stateProperties);
    } catch (Exception e) {
        throw new RuntimeException("Error while getting state", e);
    }
}

@Override
public <T> ListState<T> getListState(ListStateDescriptor<T>
stateProperties) {
    requireNonNull(stateProperties, "The state properties must not
be null");
    try {

stateProperties.initializeSerializerUnlessSet(executionConfig);
        ListState<T> originalState =
getPartitionedState(stateProperties);
        return new UserFacingListState<>(originalState);
    } catch (Exception e) {
        throw new RuntimeException("Error while getting state", e);
    }
}

private <S extends State> S getPartitionedState(StateDescriptor<S,
?> stateDescriptor) throws Exception {
    return keyedStateBackend.getPartitionedState(
            VoidNamespace.INSTANCE,
            VoidNamespaceSerializer.INSTANCE,
            stateDescriptor);
}
```

DefaultKeyedStateStore的初始化调用链为StreamTask-->invoke-->initializeState()-->initializeOperators()-->operator.initializeState()-->initKeyedState() + initOperatorState(operatorStateHandlesBackend); 想深入了解的可以顺着这条线看下去。 前文我们讲过StreamTask在其invoke方法里面初始化了keyedStateBackend和operateStatebackend。在生产环境我们配置的都是rocksDB,他们俩都是RocksDBStateBackend派生出来的。具体怎么派生的请看 `StreamTask line722,line741` 。从上面的代码可以看出创建各种state的方法最后调用的都是keyedStateBackend.getPartitionedState方法。 那么我们来看看RocksDBStateBackend的父类AbstractKeyedStateBackend对这个方法的实现：

```
//  AbstractKeyedStateBackend line372
public <N, S extends State> S getPartitionedState(
        final N namespace,
        final TypeSerializer<N> namespaceSerializer,
        final StateDescriptor<S, ?> stateDescriptor) throws
Exception {

    checkNotNull(namespace, "Namespace");

    if (lastName != null &&
lastName.equals(stateDescriptor.getName())) {
        lastState.setCurrentNamespace(namespace);
        return (S) lastState;
    }

    InternalKvState<?> previous =
keyValueStatesByName.get(stateDescriptor.getName());
    if (previous != null) {
        lastState = previous;
        lastState.setCurrentNamespace(namespace);
        lastName = stateDescriptor.getName();
        return (S) previous;
    }

    final S state = getOrCreateKeyedState(namespaceSerializer,
stateDescriptor);
    final InternalKvState<N> kvState = (InternalKvState<N>) state;

    lastName = stateDescriptor.getName();
    lastState = kvState;
    kvState.setCurrentNamespace(namespace);

    return state;
}
```

AbstractKeyedStateBackend里面也定义了获取各种state的抽象方法。上文的代码里面的getOrCreateKeyedState方法调用的就是这些抽象方法。见下面的代码：

```
// AbstractKeyedStateBackend line312
// create a new blank key/value state
    S state = stateDescriptor.bind(new StateBinder() {
        @Override
        public <T> ValueState<T>
createValueState(ValueStateDescriptor<T> stateDesc) throws
Exception {
            return
AbstractKeyedStateBackend.this.createValueState(namespaceSerializer
, stateDesc);
        }

        @Override
        public <T> ListState<T>
createListState(ListStateDescriptor<T> stateDesc) throws Exception
{
            return
AbstractKeyedStateBackend.this.createListState(namespaceSerializer,
stateDesc);
        }
```

那么谁实现了这些抽象方法呢？RocksDBStateBackend是AbstractKeyedStateBackend的子类，我们看看她是怎么实现的？我们以createListState举例：

```
// RocksDBStateBackend line1679
@Override
protected <N, T> InternalListState<N, T> createListState(
    TypeSerializer<N> namespaceSerializer,
    ListStateDescriptor<T> stateDesc) throws Exception {

    ColumnFamilyHandle columnFamily = getColumnFamily(stateDesc,
namespaceSerializer);

    return new RocksDBListState<>(columnFamily,
namespaceSerializer, stateDesc, this);
}
```

最终我们发现我们在api里面调用的state是根据statebackend的不同实现决定的。在这里是RocksDBListState。既然我们现在获取到了ListState，我们不防看看它是怎么存放数据的？以add方法为例：

```
// RocksDBListState line109
@Override
public void add(V value) throws IOException {
    try {
        writeCurrentKeyWithGroupAndNamespace();
        byte[] key = keySerializationStream.toByteArray();
        keySerializationStream.reset();
        DataOutputViewStreamWrapper out = new
DataOutputViewStreamWrapper(keySerializationStream);
        valueSerializer.serialize(value, out);
        backend.db.merge(columnFamily, writeOptions, key,
keySerializationStream.toByteArray());
    } catch (Exception e) {
        throw new RuntimeException("Error while adding data to
RocksDB", e);
    }
}
```

上面的代码backend.db就是RocksDB实例，我们add数据的时候其实调用的是RocksDB实例merge方法。最后我们终于搞清楚我们的自己的state数据是怎么存入后端的backend上面了。那么下个问题来了，backend是怎么做snapshot呢？ 带着这个问题，我们来看看RocksDBStateBackend是怎么实现的？
RocksDBKeyedStateBackend 将 用户的state 数据存储进 RocksDB自身中(首先写TM的内存)并且在做checkpoint 的时候将state写到CheckpointStreamFactory提供的输出流中。她分为两种模式，一种是增量一种是全量。如果我们讲enableIncrementalCheckpointing置为true。则开启增量模式。

```java
// RocksDBListState line343
@Override
public RunnableFuture<KeyedStateHandle> snapshot(
    final long checkpointId,
    final long timestamp,
    final CheckpointStreamFactory streamFactory,
    CheckpointOptions checkpointOptions) throws Exception {

    if (checkpointOptions.getCheckpointType() !=
CheckpointOptions.CheckpointType.SAVEPOINT &&
        enableIncrementalCheckpointing) {
        return snapshotIncrementally(checkpointId, timestamp,
streamFactory);
    } else {
        return snapshotFully(checkpointId, timestamp,
streamFactory);
    }
}
```

下面分别看看它的增量跟全量是怎么实现的?

1. 增量快照 对应的方法为 `RocksDBListState.java snapshotIncrementally line357`
2. RocksDBStateBackend 增量快照实际的快照数据包含两部分：1. meta 数据、2. state 数据。首先meta 数据直接通过 CheckpointStreamFactory 写出到快照目录。
3. state 数据的写入，会对 RocksDB 自身做一次 snapshot(对应方法 snapshotOperation.takeSnapshot())，后再将 checkpoint 写出到 CheckpointStateOutputStream，将 RocksDB 快照 写出到 CheckpointStateOutputStream 的实现细节：
4. 创建用于创建可打开快照的Checkpoint对象然后在RocksDB的base目录下新建一个chk-checkpointId的RocksDB快照目录，并将快照保存下来【实际为创建 hard link】，这部分的快照包含两种文件，sst 文件 + misc 文件，接下来就是将这部分文件写出到 CheckpointStateOutputStream
5. 拿到上次 checkpoint 的 sst 文件列表【实际为 stateHandleID -> StreamStateHandle 映射关系】
6. 遍历RocksDB的base目录下文件列表：如果是 sst 文件，会根据拿到上次 checkpoint的sst文件列表中的映射关系查询上次是否已做过快照，如果有跳过快照只将文件名【和快照句柄占位符 PlaceholderStreamStateHandle】保存下来，否则会将sst文件写出到 CheckpointStateOutputStream 并保存 文件名 -> 快照句柄 映射关系；如果是 misc 文件，直接将文件写出到 CheckpointStateOutputStream，并保存 文件名 -> 快照句柄 映射关系;

```java
//RocksDBListState line343
private RunnableFuture<KeyedStateHandle> snapshotIncrementally(
    final long checkpointId,
    final long checkpointTimestamp,
    final CheckpointStreamFactory checkpointStreamFactory) throws
Exception {

    if (db == null) {
        throw new IOException("RocksDB closed.");
    }

    if (kvStateInformation.isEmpty()) {
        if (LOG.isDebugEnabled()) {
            LOG.debug("Asynchronous RocksDB snapshot performed on
empty keyed state at " +
                checkpointTimestamp + " . Returning null.");
        }
        return DoneFuture.nullValue();
    }
    final RocksDBIncrementalSnapshotOperation<K> snapshotOperation
= new RocksDBIncrementalSnapshotOperation<>(
            this,
            checkpointStreamFactory,
            checkpointId,
            checkpointTimestamp);

    try {
        snapshotOperation.takeSnapshot();
    } catch (Exception e) {
        snapshotOperation.stop();
        snapshotOperation.releaseResources(true);
        throw e;
    }

    return new FutureTask<KeyedStateHandle>(
        () -> snapshotOperation.materializeSnapshot()
    ) {
        @Override
        public boolean cancel(boolean mayInterruptIfRunning) {
            snapshotOperation.stop();
            return super.cancel(mayInterruptIfRunning);
        }

        @Override
        protected void done() {
            snapshotOperation.releaseResources(isCancelled());
        }
    };
}
```

takeSnapshot()这个方法做了两件事：首先meta 数据直接通过 CheckpointStreamFactory 写出到快照目录。对RocksDB自身做一次 snapshot。我们里看看其实现:

```
//RocksDBListState line343
void takeSnapshot() throws Exception {

final long lastCompletedCheckpoint;

// use the last completed checkpoint as the comparison base.使用最后完成的检查点作为比较基础。
synchronized (stateBackend.materializedSstFiles) {
    lastCompletedCheckpoint =
stateBackend.lastCompletedCheckpointId;
```

```
    baseSstFiles =
stateBackend.materializedSstFiles.get(lastCompletedCheckpoint);
}

LOG.trace("Taking incremental snapshot for checkpoint {}. Snapshot
is based on last completed checkpoint {} " +
    "assuming the following (shared) files as base: {}.",
checkpointId, lastCompletedCheckpoint, baseSstFiles);

// save meta data
for (Map.Entry<String, Tuple2<ColumnFamilyHandle,
RegisteredKeyedBackendStateMetaInfo<?, ?>>> stateMetaInfoEntry
    : stateBackend.kvStateInformation.entrySet()) {

stateMetaInfoSnapshots.add(stateMetaInfoEntry.getValue().f1.snapsho
t());
}

// save state data
backupPath = new
Path(stateBackend.instanceBasePath.getAbsolutePath(), "chk-" +
checkpointId);

LOG.trace("Local RocksDB checkpoint goes to backup path {}.",
backupPath);

backupFileSystem = backupPath.getFileSystem();
if (backupFileSystem.exists(backupPath)) {
    throw new IllegalStateException("Unexpected existence of the
backup directory.");
}

// create hard links of living files in the checkpoint path 在检查点
路径中创建生活文件的硬链接，对自身做一次snapshot;
Checkpoint checkpoint = Checkpoint.create(stateBackend.db);
checkpoint.createCheckpoint(backupPath.getPath());
}
```

其实最关键的就是最下面的两句代码。创建用于创建可打开快照的Checkpoint对象，然后在同一磁盘上构建RocksDB的可打开快照，该该快照接受同一磁盘上的输出目录，并在目录下（1）指向现有实时SST文件的硬链接SST文件（2）复制的清单文件和其他文件。关于这块的知识请参考：https://www.jianshu.com/p/87a9c9be1c28 takeSnapshot()已经完成，我们看下物质化Snapshot的方法是如何实现实现的？对应方法为：materializeSnapshot()

```
KeyedStateHandle materializeSnapshot() throws Exception {
```

```
KeyedStateHandle materializeSnapshot() throws Exception {

stateBackend.cancelStreamRegistry.registerCloseable(closeableRegist
ry);
        // write meta data
        metaStateHandle = materializeMetaData();
        // write state data 写入元数据

Preconditions.checkState(backupFileSystem.exists(backupPath));
        FileStatus[] fileStatuses =
backupFileSystem.listStatus(backupPath);
        if (fileStatuses != null) {
            for (FileStatus fileStatus : fileStatuses) {
                final Path filePath = fileStatus.getPath();
                final String fileName = filePath.getName();
                final StateHandleID stateHandleID = new
StateHandleID(fileName);
                if (fileName.endsWith(SST_FILE_SUFFIX)) {
                    final boolean existsAlready =  baseSstFiles !=
null && baseSstFiles.contains(stateHandleID);
                    if (existsAlready) {
                        // we introduce a placeholder state handle,
that is replaced with the
                        // 我们引入了一个占位符状态句柄，它被替换为共享状态
注册表中的原始句柄（从以前的检查点创建）
                        // original from the shared state registry
(created from a previous checkpoint)
                        sstFiles.put(stateHandleID,new
PlaceholderStreamStateHandle());
                    } else {
                        sstFiles.put(stateHandleID,
materializeStateData(filePath));
                    }
                } else {
                    StreamStateHandle fileHandle =
materializeStateData(filePath);
                    miscFiles.put(stateHandleID, fileHandle);
                }
            }
        }
        synchronized (stateBackend.materializedSstFiles) {
            stateBackend.materializedSstFiles.put(checkpointId,
sstFiles.keySet());
        }
        return new IncrementalKeyedStateHandle(
            stateBackend.backendUID,
            stateBackend.keyGroupRange,
            checkpointId,
            sstFiles,
            miscFiles,
            metaStateHandle);
    }
```

首先这个方法主要干了两件事，1是物质化meta数据、二是物质化state数据。backupPath 在 takeSnapshot() 方法初始化 `backupPath = new Path(stateBackend.instanceBasePath.getAbsolutePath(), "chk-" + checkpointId);` 然后循环便利下这个路基下的文件， 如果是以.sst结尾的文件，则判断文件是否已经存在过，如果已经存在了跳过快照，只将文件名【和快照句柄占位符 PlaceholderStreamStateHandle】保存下来，否则会将 sst 文件写出到 CheckpointStateOutputStream 并保存 文件名 -> 快照句柄 映射关系；以上每个文件.sst都要读取内容再写出到一个单独的输出文件【占位符除外】; 总体来说增量备份只是减少为文件的写入。

下面我们来看看全量快照是怎么做的?

全量快照,对应的方法为 `RocksDBListState.java snapshotFully line404` RocksDBStateBackend 全量快照的整体思路是通过 CheckpointStreamFactory 拿到 CheckpointStateOutputStream 并将 RocksDB 自身的 snapshot 全量写出，首先也是拿到RocksDB自身的snapshot对象，对应代码 `snapshotOperation.takeDBSnapShot(checkpointId, timestamp);` 这个方法返回到当前的DB状态的句柄。使用此句柄创建的迭代器将创建当前DB状态的稳定快照的视图。 当不再需要快照时，调用者必须调用ReleaseSnapshot（result） 。 然后通过 CheckpointStreamFactory 拿到 CheckpointStateOutputStream 作为快照写出流，分别将快照的 meta 信息和数据写到 CheckpointStateOutputStream;拿到CheckpointStateOutputStream输出流的句柄， 获得状态 offset，将 k-v 数据读依次写入,以上只写一个输出文件;

```
//RocksDBListState.java line404
// implementation of the async IO operation, based on FutureTask
AbstractAsyncCallableWithResources<KeyedStateHandle> ioCallable =
    new AbstractAsyncCallableWithResources<KeyedStateHandle>() {
        @Override
        protected void acquireResources() throws Exception {

cancelStreamRegistry.registerCloseable(snapshotCloseableRegistry);
            snapshotOperation.openCheckpointStream();
        }

        @Override
        protected void releaseResources() throws Exception {
            closeLocalRegistry();
            releaseSnapshotOperationResources();
        }

        private void releaseSnapshotOperationResources() {
            // hold the db lock while operation on the db to guard
us against async db disposal
            snapshotOperation.releaseSnapshotResources();
        }

        @Override
        protected void stopOperation() throws Exception {
            closeLocalRegistry();
        }

        private void closeLocalRegistry() {
            if
(cancelStreamRegistry.unregisterCloseable(snapshotCloseableRegistry
)) {
                try {
                    snapshotCloseableRegistry.close();
                } catch (Exception ex) {
                    LOG.warn("Error closing local registry", ex);
                }
            }
        }
        @Override
        public KeyGroupsStateHandle performOperation() throws
Exception {
            long startTime = System.currentTimeMillis();

            if (isStopped()) {
                throw new IOException("RocksDB closed.");
            }

            snapshotOperation.writeDBSnapshot();

            LOG.info("Asynchronous RocksDB snapshot ({},
asynchronous part) in thread {} took {} ms.",
                streamFactory, Thread.currentThread(),
(System.currentTimeMillis() - startTime));

            return
snapshotOperation.getSnapshotResultStateHandle();
        }
    };

LOG.info("Asynchronous RocksDB snapshot (" + streamFactory + ",
synchronous part) in thread " +
    Thread.currentThread() + " took " + (System.currentTimeMillis()
- startTime) + " ms.");

return AsyncStoppableTaskWithCallback.from(ioCallable);
```

上述代码最主要的就是`snapshotOperation.writeDBSnapshot();`这代表着开始保存Snapshot；也是写两部分的数据一部分是meta数据，一部分是state数据。至此state的写入过程结束了！