check point检查点

声明

以下仅基于1.14.2版本

概念

检查点允许Flink恢复数据流中的状态和位置。

Flink定期为每个算子的所有状态进行持久化快照，并将这些快照复制到更持久的地方；如果发生故障，Flink可以恢复应用程序的完整状态。

这些快照的存储位置是通过检查点存储定义的，其中有两种检查点存储的实现：

- *FileSystemCheckpointStorage*，文件系统，以file://或hdfs://指定
- *JobManagerCheckpointStorage*，jobManager的jvm

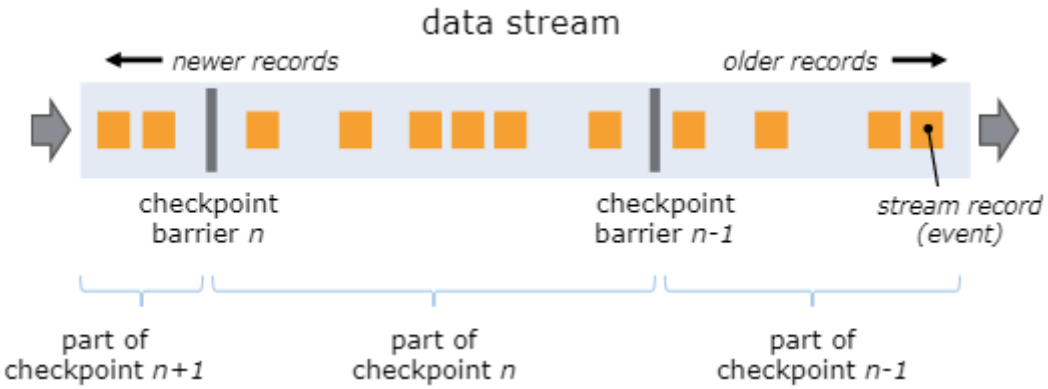
值得注意的一点是，有时候没有用户自定义的state，但是检查点仍然会存储一些state数据，这些数据来自于flink内部封装的函数，例如reduce。

原理

Flink基于以下条件采用轻量级分布式快照实现应用容错：

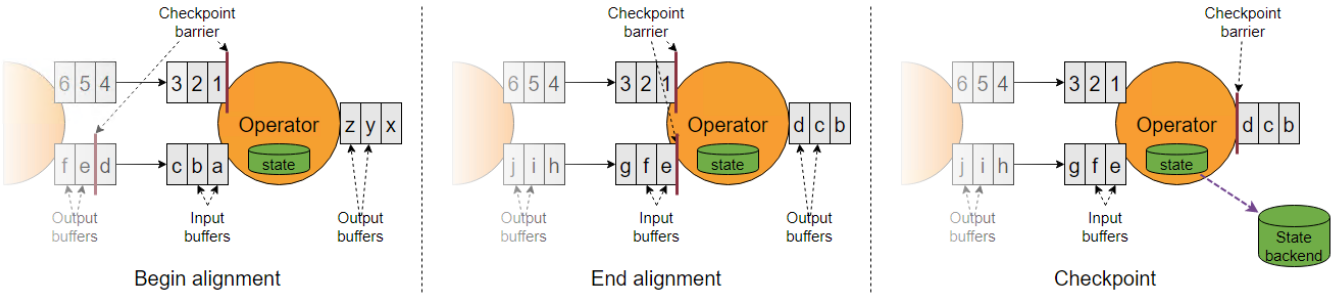
- job的异常极少发生，每次发生后需停付出极大代价回滚
- 为了降低延迟，快照需要短时间内完成
- Task是不会变动的，从快照恢复不会修改task的并行度

CheckpointCoordinator周期性向流中发送Barrier来切分数据流；每个算子接收到一个barrier时，暂停数据处理，将当前状态制作成快照，保存到指定的持久化存储中，最后向CheckpointCoordinator报告，同时向自身所有下游算子广播该barrier，完成后继续处理数据。

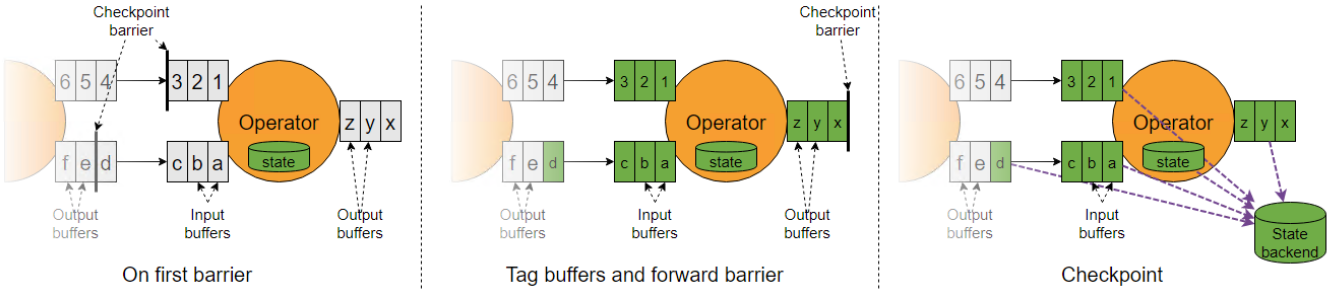


当CheckpointCoordinator在规定时间内收到所有的算子报告时，认为周期快照成功。

多输入流检查点对齐：



多输入流检查点非对齐(不成熟，但是也能用)：



有一点需要注意的是，对于对齐的检查点，当多并行流数据倾斜过大时，很容易造成检查点超时，因为需要对齐。

使用

默认情况下，检查点是禁用的(1.14)，启用检查点的代码：

```
    try {
        Configuration configuration = new Configuration();
        configuration.setInteger(RestOptions.PORT, 8082);

        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment(configuration);
        env.setParallelism(1);
        // 开启检查点，同时配置检查点间隔为5000ms
        env.enableCheckpointing(5000L);
        // 设置检查点目录存储目录为文件系统
        env.getCheckpointConfig().setCheckpointStorage(new
Path("file:///C://untitled"));

        env.addSource(new SourceFunction<Long>() {
            @Override
            public void run(SourceContext<Long> ctx) throws Exception {
                while (true) {
                    Thread.sleep(1000L);
                    ctx.collect(1L);
                }
            }

            @Override
            public void cancel() {
```

```
            }
        }).map(i -> i).print();

        env.execute();
    } catch (Exception e) {
        e.printStackTrace();
    }
```

检查点的属性

```
// 检查点配置的修改方式为:
// env.getCheckpointConfig()后添加配置
public class CheckpointConfig {
    //默认检查点类型是精确一次
    public static final CheckpointingMode DEFAULT_MODE =
CheckpointingMode.EXACTLY_ONCE;

    //默认检查点超时时间10min
    public static final long DEFAULT_TIMEOUT = 10 * 60 * 1000;

    //默认检查点之间最短间隔0
    public static final long DEFAULT_MIN_PAUSE_BETWEEN_CHECKPOINTS = 0;

    //默认最大同时生效检查点数量1
    public static final int DEFAULT_MAX_CONCURRENT_CHECKPOINTS = 1;

    //默认不容忍检查点失败
    public static final int UNDEFINED_TOLERABLE_CHECKPOINT_NUMBER = -1;

    //检查点默认id，执行检查点时忽略正在运行的数据
    public static final int DEFAULT_CHECKPOINT_ID_OF_IGNORED_IN_FLIGHT_DATA = -1;

    // 默认检查点间隔-1，即不支持周期检查点
    private long checkpointInterval = -1;

    // 默认为非强制检查点
    private boolean forceCheckpointing;

    // 默认不支持非对齐检查点
    private boolean forceUnalignedCheckpoints;

    // 默认不支持非对齐检查点
    private boolean unalignedCheckpointsEnabled;
}

// CheckpointProperties由CheckpointConfig生成，具体的生成代码是
// CheckpointProperties.forCheckpoint(chkConfig.getCheckpointRetentionPolicy())
public class CheckpointProperties {
    // Type - checkpoint / savepoint.
    private final CheckpointType checkpointType;
    // whether the checkpoint should be forced.
```

```
    // 一个强制检查点不被最大并行数和最短间隔影响，应用场景未知
    // 代码里看，一个非对齐检查点应该forced
    private final boolean forced;
    // 暂不考虑
    private final boolean discardSubsumed;
    // 暂不考虑
    private final boolean discardFinished;
    // 暂不考虑
    private final boolean discardCancelled;
    // 暂不考虑
    private final boolean discardFailed;
    // 暂不考虑
    private final boolean discardSuspended;
}
```

实际业务中开启检查点的代码逻辑为：

```
class StreamExecutionEnvironment {
    // interval - 两次检查点之间的间隔(毫秒)
    public StreamExecutionEnvironment enableCheckpointing(long interval) {
        checkpointCfg.setCheckpointInterval(interval);
        return this;
    }

    public StreamExecutionEnvironment enableCheckpointing(long interval,
CheckpointingMode mode) {
        checkpointCfg.setCheckpointingMode(mode);
        checkpointCfg.setCheckpointInterval(interval);
        return this;
    }
}

enum CheckpointingMode {
    // 每个元素都会在算子中精准出现一次，多分区的情况下，会延迟；需要对齐分区数据，需要维
持高吞吐
    EXACTLY_ONCE,
    // 延迟较低，可能导致重复数据
    AT_LEAST_ONCE
}

// 用户无法在代码中指定，所以默认是NEVER_RETAIN_AFTER_TERMINATION
// 可以由ExternalizedCheckpointCleanup指定生成，对应关系为：
// RETAIN_ON_CANCELLATION -- RETAIN_ON_CANCELLATION
// RETAIN_ON_FAILURE -- DELETE_ON_CANCELLATION
enum CheckpointRetentionPolicy {
    /** Checkpoints should be retained on cancellation and failure. */
    /** 检查点在job取消或失败后保留 */
    RETAIN_ON_CANCELLATION,

    /** Checkpoints should be retained on failure, but not on cancellation. */
    /** 检查点仅在job失败后保留，job取消时删除 */
    RETAIN_ON_FAILURE,
```

```
    /** Checkpoints should always be cleaned up when an application reaches a
terminal state. */
    /** job结束后，清除所有检查点 */
    NEVER_RETAIN_AFTER_TERMINATION
}

// 外部检查点的开启代码为：
env.getCheckpointConfig().enableExternalizedCheckpoints(ExternalizedCheckpointClea
nup);
enum ExternalizedCheckpointCleanup {
         /**
         * Delete externalized checkpoints on job cancellation.
         * 当job被取消时删除
         */
        DELETE_ON_CANCELLATION(true),

        /**
         * Retain externalized checkpoints on job cancellation.
         * 当job被取消时保留
         */
        RETAIN_ON_CANCELLATION(false);

        // 是否删除
        private final boolean deleteOnCancellation;
}
```

Checkpointing的校验：

```
public class StreamJobGraphGenerator{
    //StreamJobGraphGenerator.java
    private void configureCheckpointing() {
        CheckpointConfig cfg = streamGraph.getCheckpointConfig();

        // checkpoint 间隔校验
        long interval = cfg.getCheckpointInterval();
        if (interval < MINIMAL_CHECKPOINT_TIME) {
            // interval < 10ms 被认为是禁止检查点
            // interval == Long.MAX_VALUE时被认为是禁止周期性检查点
            interval = Long.MAX_VALUE;
        }

        // 枚举类CheckpointRetentionPolicy
        // 提供job终止后的检查点策略
        CheckpointRetentionPolicy retentionAfterTermination;
        // 此处的判断指的是checkpointConfig里的externalizedCheckpointCleanup !=
null
        // externalizedCheckpointCleanup字段指的是外部检查点的清除策略
        // 外部检查点开启必须指定清除策略，所以可以根据是否存在策略判断是否开启
        if (cfg.isExternalizedCheckpointsEnabled()) {
            CheckpointConfig.ExternalizedCheckpointCleanup cleanup =
                    cfg.getExternalizedCheckpointCleanup();
```

```
            // Sanity check
            if (cleanup == null) {
                throw new IllegalStateException(
                        "Externalized checkpoints enabled, but no cleanup mode
configured.");
            }
            // 当用户配置了job取消时删除外部检查点，则提供RETAIN_ON_FAILURE
            // 当用户配置了job取消时保留外部检查点，则提供RETAIN_ON_CANCELLATION
            retentionAfterTermination =
                    cleanup.deleteOnCancellation()
                            ? CheckpointRetentionPolicy.RETAIN_ON_FAILURE
                            : CheckpointRetentionPolicy.RETAIN_ON_CANCELLATION;
        } else {
            // 当用户不开启外部检查点时，默认提供NEVER_RETAIN_AFTER_TERMINATION
            retentionAfterTermination =
CheckpointRetentionPolicy.NEVER_RETAIN_AFTER_TERMINATION;
        }

        // 在正常情况下，hooks列表为空，实现WithMasterCheckpointHook接口的算子只有
ExternallyInducedSource
        // ExternallyInducedSource算子不会响应检查点协调器发出的检查点消息，只会在接受
到某些数据是触发检查点，基本上用不到。
        // 目前来看，自定义触发检查点的函数只有checkpoint id做参数，还有待研究。
        // MasterTriggerRestoreHook是在检查点协调器触发或恢复检查点时使用的，内部接口
Factory用于创建一个具体的hook实例
        // MasterTriggerRestoreHook中包括triggerCheckpoint(long checkpointId, long
timestamp, Executor executor)方法
        // 该方法被检查点协调器调用，将触发检查点发送给数据源。
        final ArrayList<MasterTriggerRestoreHook.Factory> hooks = new ArrayList<>
();
        for (StreamNode node : streamGraph.getStreamNodes()) {
            if (node.getOperatorFactory() instanceof UdfStreamOperatorFactory) {
                Function f =
                        ((UdfStreamOperatorFactory)
node.getOperatorFactory()).getUserFunction();

                if (f instanceof WithMasterCheckpointHook) {
                    hooks.add(
                            new FunctionMasterCheckpointHookFactory(
                                    (WithMasterCheckpointHook<?>) f));
                }
            }
        }

        // because the hooks can have user-defined code, they need to be stored as
        // eagerly serialized values
        // 将列表转化为可执行序列化的数组
        // 正常使用来说，列表为空
        final SerializedValue<MasterTriggerRestoreHook.Factory[]> serializedHooks;
        if (hooks.isEmpty()) {
            serializedHooks = null;
        } else {
            try {
                MasterTriggerRestoreHook.Factory[] asArray =
```

```
                        hooks.toArray(new
MasterTriggerRestoreHook.Factory[hooks.size()]);
                serializedHooks = new SerializedValue<>(asArray);
            } catch (IOException e) {
                throw new FlinkRuntimeException("Trigger/restore hook is not
serializable", e);
            }
        }

        // 将状态后端转化为可执行序列化数组
        final SerializedValue<StateBackend> serializedStateBackend;
        if (streamGraph.getStateBackend() == null) {
            serializedStateBackend = null;
        } else {
            try {
                serializedStateBackend =
                        new SerializedValue<StateBackend>
(streamGraph.getStateBackend());
            } catch (IOException e) {
                throw new FlinkRuntimeException("State backend is not
serializable", e);
            }
        }

        // 将存储空间转化为可执行序列化数组
        final SerializedValue<CheckpointStorage> serializedCheckpointStorage;
        if (streamGraph.getCheckpointStorage() == null) {
            serializedCheckpointStorage = null;
        } else {
            try {
                serializedCheckpointStorage =
                        new SerializedValue<>(streamGraph.getCheckpointStorage());
            } catch (IOException e) {
                throw new FlinkRuntimeException("Checkpoint storage is not
serializable", e);
            }
        }

        // 生成JobCheckpointingSettings并添加进jobGraph
    }
}
```

周期性检查点的生成：

```
// 当job状态变动时，会执行jobStatusChanges()
// 该监听器会启动或关闭周期性检查点
public class CheckpointCoordinatorDeActivator implements JobStatusListener {
    // 检查器协调器，后续会具体说明
    // 检查器协调器会协调算子和状态的分布式快照，协调器会通过向相关任务发送消息来触发检查
点
    private final CheckpointCoordinator coordinator;
```

```
    <init>

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

public class CheckpointCoordinator {
    // 标志下一次检查点是否正常执行，每次发送前都会判断
    private boolean periodicScheduling;
    ...
    public void startCheckpointScheduler() {
        synchronized (lock) {
            if (shutdown) {
                throw new IllegalArgumentException("Checkpoint coordinator is shut
down");
            }
            Preconditions.checkState(
                    // 这一步会判断baseInterval != Long.MAX_VALUE，即
CheckpointConfig.checkpointInterval，表明不支持周期性检查点
                    // 再往前追溯，即配置检查点周期小于10ms时，被flink认为interval =
Long.MAX_VALUE
                    isPeriodicCheckpointingConfigured(),
                    "Can not start checkpoint scheduler, if no periodic
checkpointing is configured");

            // make sure all prior timers are cancelled
            stopCheckpointScheduler();

            // stopCheckpointScheduler()就是反过来
            periodicScheduling = true;
            // 初始化时随机一个等待时间，随机时间为[minPauseBetweenCheckpoints,
baseInterval + 1L}
            currentPeriodicTrigger =
scheduleTriggerWithDelay(getRandomInitDelay());
        }
    }

    private ScheduledFuture<?> scheduleTriggerWithDelay(long initDelay) {
        // ScheduledExecutorServiceAdapter
        // 经过初始的等待时间(initDelay)后，每隔一段时间(baseInterval)就执行一次
        return timer.scheduleAtFixedRate(
                new ScheduledTrigger(), initDelay, baseInterval,
TimeUnit.MILLISECONDS);
    }
```

```
    public void stopCheckpointScheduler() {
        synchronized (lock) {
            periodicScheduling = false;

            // 取消当前的周期触发任务
            cancelPeriodicTrigger();

            // 填充报错信息Checkpoint Coordinator is suspending.
            final CheckpointException reason =
                    new
CheckpointException(CheckpointFailureReason.CHECKPOINT_COORDINATOR_SUSPEND);
            // 将队列中下一个检查点请求抛出
            // 将已下发的pending checkpoint释放
            abortPendingAndQueuedCheckpoints(reason);

            // 将失败的触发次数置为0
            numUnsuccessfulCheckpointsTriggers.set(0);
        }
    }

    private void cancelPeriodicTrigger() {
        if (currentPeriodicTrigger != null) {
            // 允许当前检查点执行完毕
            currentPeriodicTrigger.cancel(false);
            currentPeriodicTrigger = null;
        }
    }

    private long getRandomInitDelay() {
        // 返回[minPauseBetweenCheckpoints, baseInterval + 1L}的随机数
        // 即CheckpointConfig.minPauseBetweenCheckpoints, 到
CheckpointConfig.checkpointInterval
        return ThreadLocalRandom.current().nextLong(minPauseBetweenCheckpoints,
baseInterval + 1L);
    }
    ...
}
```

关键组件介绍

检查点 Checkpoint

> A checkpoint, pending or completed.

```
public interface Checkpoint {
    long getCheckpointID();

    void discard() throws Exception;
}
```

检查点执行计划 CheckpointPlan

```
// 某次检查点的计划，包括哪些task需要被(检查点协调器)触发检查点，哪些task在等待接受检查点，哪些task需要提交检查点结果
// The plan of one checkpoint, indicating which tasks to trigger, waiting for acknowledge or commit for one specific checkpoint.
public interface CheckpointPlan {
    // 获取需要被触发的task，指source function
    List<Execution> getTasksToTrigger();
    // 获取需要接受检查点信息的task
    List<Execution> getTasksToWaitFor();
    // 获取检查点时仍然在运行的task，这些task需要在检查点确认时发送消息
    List<ExecutionVertex> getTasksToCommitTo();
    // 获取已经完成的task
    List<Execution> getFinishedTasks();
    ...
}
```

计算下一个检查点执行时需要的检查点计划

```
// CheckpointPlanCalculator有唯一实现类DefaultCheckpointPlanCalculator
public interface CheckpointPlanCalculator {
    // 计算下一个检查点执行时需要的检查点计划
    CompletableFuture<CheckpointPlan> calculateCheckpointPlan();
}

public class DefaultCheckpointPlanCalculator {
    ...
    @Override
    // 计算下一个检查点执行时需要的检查点计划
    public CompletableFuture<CheckpointPlan> calculateCheckpointPlan() {
        return CompletableFuture.supplyAsync(
                () -> {
                    try {
                        // 检查是否存在已完成的task且环境配置是否支持在已完成的task后执行检查点
                        // 注：这个配置是默认false的，需要的话可以配置execution.checkpointing.checkpoints-after-tasks-finish.enabled
                        if (context.hasFinishedTasks() && !allowCheckpointsAfterTasksFinished) {
                            throw new CheckpointException("Some tasks of the job have already finished and checkpointing with finished tasks is not enabled.",CheckpointFailureReason.NOT_ALL_REQUIRED_TASKS_RUNNING);
                        }

                        // 检查是否全部task正在运行，若有task不在运行，则抛出异常
                        checkAllTasksInitiated();

                        CheckpointPlan result =
                                context.hasFinishedTasks()
```

```
                                        // 对于NGCV当前场景，不需要考虑finish的情况
                                        ? calculateAfterTasksFinished()
                                        // 需要触发检查点的是全部的sourceFunction，
需要等待的是全部task
                                        // 生成一个默认检查点计划，其中：
                                        // trigger列表指全部source function
                                        // waitFor列表指全部task
                                        // commitTo列表指全部task
                                        // 其余列表为空
                                        // Computes the checkpoint plan when all
tasks are running.
                                        // It would simply marks all the source
tasks as need to trigger and all the tasks as need to wait and commit.
                                        : calculateWithAllTasksRunning();

                        // 检查是否全部的wait列表状态为running
                        checkTasksStarted(result.getTasksToWaitFor());

                        return result;
                    } catch (Throwable throwable) {
                        throw new CompletionException(throwable);
                    }
                },
                context.getMainExecutor());
    }
    ...
}
```

完成的检查点 CompletedCheckpoint

一个CompletedCheckpoint描述了某个检查点最终的成功状态，这个检查点被认为是成功的。

```
public class CompletedCheckpoint implements Serializable, Checkpoint {
    // 当前job的id
    private final JobID job;
    // 当前检查点的id
    private final long checkpointID;
    // 当前检查点触发时的时间戳
    private final long timestamp;
    // 当前检查点结束时的时间戳
    private final long completionTimestamp;
    // 记录不同的算子和算子状态
    private final Map<OperatorID, OperatorState> operatorStates;
    // 当前检查点的配置
    private final CheckpointProperties props;
    // 检查点协调器生成的状态，做什么有什么用未知
    private final Collection<MasterState> masterHookStates;
    // 检查点的存储未知
    private final CompletedCheckpointStorageLocation storageLocation;
    // 和外部检查点相关
    private final StreamStateHandle metadataHandle;
```

```
    // 指向外部检查点的指针，例如文件路径
    private final String externalPointer;
    // 某个回调函数，干什么未知
    @Nullable private transient volatile CompletedCheckpointStats.DiscardCallback
discardCallback;
    ...
}
```

检查点触发请求 CheckpointTriggerRequest

```
class CheckpointTriggerRequest {
    // 生成请求的时间
    final long timestamp;
    // 当前检查点配置
    final CheckpointProperties props;e
    final @Nullable String externalSavepointLocation;
    final boolean isPeriodic;
    private final CompletableFuture<CompletedCheckpoint> onCompletionPromise = new
CompletableFuture<>();
}
```

检查点请求选择器 CheckpointRequestDecider

```
public class CheckpointRequestDecider {
    private static final int LOG_TIME_IN_QUEUE_THRESHOLD_MS = 100;
    private static final int DEFAULT_MAX_QUEUED_REQUESTS = 1000;

    private final int maxConcurrentCheckpointAttempts;
    private final Consumer<Long> rescheduleTrigger;
    private final Clock clock;
    private final long minPauseBetweenCheckpoints;
    private final Supplier<Integer> pendingCheckpointsSizeSupplier;
    private final Supplier<Integer> numberOfCleaningCheckpointsSupplier;
    // 优先顺序：非保存点、非强制、周期性、请求生成时间、哈希值
    private final NavigableSet<CheckpointTriggerRequest> queuedRequests =
            new TreeSet<>(checkpointTriggerRequestsComparator());
    // default(all) is 1000
    private final int maxQueuedRequests;

    // param newRequest：检查点协调器生成的某个检查点请求
    // param isTriggering：当前是否正在执行检查点，检查点协调器中存储的字段值
    // param lastCompletionMs：上一次成功执行的检查点的相对时间
    // 提交一个检查点请求，并决定执行当前请求或队列中其他请求
    // return 一个应该被执行的检查点
    Optional<CheckpointTriggerRequest> chooseRequestToExecute(
            CheckpointTriggerRequest newRequest, boolean isTriggering, long
lastCompletionMs) {
        // 当队列已满且队列优先级最高的不是周期任务时，丢弃当前请求
        if (queuedRequests.size() >= maxQueuedRequests &&
```

```
!queuedRequests.last().isPeriodic) {
            // there are only non-periodic (ie user-submitted) requests enqueued -
retain them and drop the new one
            // 报错信息：The maximum number of queued checkpoint requests exceeded
            newRequest.completeExceptionally(new
CheckpointException(TOO_MANY_CHECKPOINT_REQUESTS));
            return Optional.empty();
        } else {
            // 将当前请求进入队列
            queuedRequests.add(newRequest);
            // 正常情况下，应该不会出现检查点过多的情况
            if (queuedRequests.size() > maxQueuedRequests) {
                // drop the last (最低优先级)
                queuedRequests
                        .pollLast()
                        .completeExceptionally(
                                new
CheckpointException(TOO_MANY_CHECKPOINT_REQUESTS));
            }

            // 选择下一个请求执行，此处不贴代码，大概是从队列中选择优先级最低的
            // 当选择的请求非强制、周期性时，检查距离上次执行是否超过间隔
            Optional<CheckpointTriggerRequest> request =
                    chooseRequestToExecute(isTriggering, lastCompletionMs);
            request.ifPresent(CheckpointRequestDecider::logInQueueTime);
            return request;
        }
    }


}
```

执行中检查点 PendingCheckpoint

一个检查点已经开始，但是还未被全部task接受。一旦全部task接受该检查点，就会转变为CompletedCheckpoint。

```
public class PendingCheckpoint {
    // 维护一个不同task的映射，表示已经接收到返回消息的task
    private final Set<ExecutionAttemptID> acknowledgedTasks;
    // 维护一个不同task的映射，表示没接收到的task
    // 构造方法中创建，将waitFor队列中的task全部写入
    // 当某个task返回message到检查点协调器时，将该task从map中移除
    private final Map<ExecutionAttemptID, ExecutionVertex>
notYetAcknowledgedTasks;
    // 标志位，代表该检查点是否被废弃
    private boolean disposed;

    // 检查点协调器接收到task返回的消息并进入该方法
    public TaskAcknowledgeResult acknowledgeTask(
            ExecutionAttemptID executionAttemptId,
```

```
            TaskStateSnapshot operatorSubtaskStates,
            CheckpointMetrics metrics,
            @Nullable PendingCheckpointStats statsCallback) {
        // 逻辑见下方枚举类
        ...
    }
}

// Result of the acknowledgedTasks method.
enum TaskAcknowledgeResult {
    // successful acknowledge of the task
    SUCCESS,
    // acknowledge message is a duplicate
    // 当notYetAcknowledgedTasks重复移除某个task时返回
    DUPLICATE,
    // unknown task acknowledged
    // 当notYetAcknowledgedTasks不存在且从未移除某个task时返回
    UNKNOWN,
    // pending checkpoint has been discarded
    // 代表当前检查点被废弃，每次task通知检查点协调器时，根据标志位disposed判断
    DISCARDED
}
```

检查点元数据 CheckpointMetaData

封装检查点的全部元数据

```
public class CheckpointMetaData {
    // id
    private final long checkpointId;
    // 检查点被触发的时间
    private final long timestamp;
    // 某个task接收到该检查点的时间
    private final long receiveTimestamp;
}
```

检查点屏障 CheckpointBarrier

检查点屏障被用于对齐检查点，这些barrier从source task中被发送出来。当barrier被普通算子接受后，就会等待当前算子全部的input(subTask)接收到Barrie；当全部的input接收到，就会执行检查点。

```
public class CheckpointBarrier {
    private final long id;
    private final long timestamp;
    private final CheckpointOptions checkpointOptions;
}
```

检查点配置 CheckpointOptions

不同的CheckpointBarrier可能有不同的配置

```
public class CheckpointOptions {
    private final CheckpointType checkpointType;
    private final CheckpointStorageLocationReference targetLocation;
    private final AlignmentType alignmentType;
    private final long alignedCheckpointTimeout;
}

// The type of checkpoint to perform
enum CheckpointType {
    // a checkpoint
    CHECKPOINT,
    // a savepoint
    SAVEPOINT,
    // A savepoint taken while suspending the job.
    SAVEPOINT_SUSPEND,
    // A savepoint taken while terminating the job.
    SAVEPOINT_TERMINATE
}

// How a checkpoint should be aligned.
enum AlignmentType {
    AT_LEAST_ONCE,
    ALIGNED,
    UNALIGNED,
    FORCED_ALIGNED
}
```

有关CheckpointType以及部分配置，和CheckpointRetentionPolicy有关，映射关系如下：

```
public class CheckpointProperties {
    /**
     * Creates the checkpoint properties for a checkpoint.
     *
     * @return Checkpoint properties for an external checkpoint.
     */
    public static CheckpointProperties forCheckpoint(
            CheckpointRetentionPolicy policy) {
        switch (policy) {
            case NEVER_RETAIN_AFTER_TERMINATION:
                return CHECKPOINT_NEVER_RETAINED;
            case RETAIN_ON_FAILURE:
                return CHECKPOINT_RETAINED_ON_FAILURE;
            case RETAIN_ON_CANCELLATION:
                return CHECKPOINT_RETAINED_ON_CANCELLATION;
            default:
                throw new IllegalArgumentException("unknown policy: " + policy);
        }
    }
```

```
    private static final CheckpointProperties CHECKPOINT_NEVER_RETAINED =
            new CheckpointProperties(
                    false, // not forced
                    CheckpointType.CHECKPOINT,
                    true,
                    true, // Delete on success
                    true, // Delete on cancellation
                    true, // Delete on failure
                    true); // Delete on suspension

    private static final CheckpointProperties CHECKPOINT_RETAINED_ON_FAILURE =
            new CheckpointProperties(
                    false, // not forced
                    CheckpointType.CHECKPOINT,
                    true,
                    true, // Delete on success
                    true, // Delete on cancellation
                    false, // Retain on failure
                    true); // Delete on suspension

    private static final CheckpointProperties CHECKPOINT_RETAINED_ON_CANCELLATION
=
            new CheckpointProperties(
                    false, // not forced
                    CheckpointType.CHECKPOINT,
                    true,
                    true, // Delete on success
                    false, // Retain on cancellation
                    false, // Retain on failure
                    false); // Retain on suspension
}
```

快照策略 SnapshotStrategy

不同的状态后端采取不同的快照策略，因为会被多个流调用，所以需要保证线程安全。

```
// @param <S> 返回的状态类型，表示快照的结果
// @param <SR> type of produced resources in the synchronous part.
public interface SnapshotStrategy<S extends StateObject, SR extends
SnapshotResources> {
    // 生成一些必要的信息，比如注册的算子状态、广播状态
    SR syncPrepareResources(long checkpointId) throws Exception;
    // 将快照写入给定的Factory，该工厂用于将状态写入流。
    SnapshotResultSupplier<S> asyncSnapshot(
            SR syncPartResource,
            long checkpointId,
            long timestamp,
            @Nonnull CheckpointStreamFactory streamFactory,
            @Nonnull CheckpointOptions checkpointOptions);
}
```

快照策略有默认实现类：DefaultOperatorStateBackendSnapshotStrategy。 快照策略调用方式：SnapshotStrategyRunner#snapshot，先调用syncPrepareResources，后asyncSnapshot SnapshotStrategyRunner被状态后端调用。 状态后端被StateHandler调用。

检查点协调器 CheckpointCoordinator

检查点协调器通过向相关任务发送消息来触发检查点并收集检查点确认消息。

```
public class CheckpointCoordinator {
    ...
    // 周期性检查点的具体任务命令
    // 经过初始的等待时间(initDelay)后，每隔一段时间(baseInterval)就执行一次
    private final class ScheduledTrigger implements Runnable {
        @Override
        public void run() {
            try {
                triggerCheckpoint(true);
            } catch (Exception e) {
                LOG.error("Exception while triggering checkpoint for job {}.",
job, e);
            }
        }
    }

    // 触发一个新的检查点并将给定的时间戳用作检查点的时间戳
    // The return value is a future. It completes when the checkpoint triggered
finishes or an error occurred.
    // isPeriodic - 该触发器是否是周期性的标志，目前全部是true
    public CompletableFuture<CompletedCheckpoint> triggerCheckpoint(boolean
isPeriodic) {
        // 提交本次checkpoint请求
        return triggerCheckpoint(checkpointProperties, null, isPeriodic);
    }

    // param props
    // param externalSavepointLocation: in periodic checkpoint is always null
    // param isPeriodic: in periodic checkpoint is always true
    public CompletableFuture<CompletedCheckpoint> triggerCheckpoint(
            CheckpointProperties props,
            @Nullable String externalSavepointLocation,
            boolean isPeriodic) {

        // 根据某种规则过滤某些检查点，原因未知
        // 因为理论上是不可能出现TERMINATE的checkpoint的
        // 只有savepoint才可能具备TERMINATE
        if (props.getCheckpointType().getPostCheckpointAction() ==
PostCheckpointAction.TERMINATE
                && !(props.isSynchronous() && props.isSavepoint())) {
            return FutureUtils.completedExceptionally(
                    new IllegalArgumentException(
                            "Only synchronous savepoints are allowed to advance
```

```
the watermark to MAX."));
        }

        // 生成一个请求
        CheckpointTriggerRequest request =
                new CheckpointTriggerRequest(props, externalSavepointLocation,
isPeriodic);

        // 选择一个请求执行

chooseRequestToExecute(request).ifPresent(this::startTriggeringCheckpoint);

        return request.onCompletionPromise;
    }

    private Optional<CheckpointTriggerRequest> chooseRequestToExecute(
            CheckpointTriggerRequest request) {
        synchronized (lock) {
            // 决定一个检查点请求是否需要被执行或被丢弃或延后
            // 具体代码见上文请求选择器
            return requestDecider.chooseRequestToExecute(
                    request, isTriggering, lastCheckpointCompletionRelativeTime);
        }
    }

    //真正的checkpoint执行逻辑
    private void startTriggeringCheckpoint(CheckpointTriggerRequest request) {
        try {
            synchronized (lock) {
                // 检查该请求是否是否是周期性的
                preCheckGlobalState(request.isPeriodic);
            }

            // 检查是否有正在执行的检查点任务
            Preconditions.checkState(!isTriggering);
            // 将标志位记为当前存在检查点正在执行
            isTriggering = true;

            final long timestamp = System.currentTimeMillis();
            // 获得检查点执行计划，详细见上文
            CompletableFuture<CheckpointPlan> checkpointPlanFuture =
                    checkpointPlanCalculator.calculateCheckpointPlan();

            // PendingCheckpoint指的是一个检查点已经开始，但是还未被全部task接受
            // 当PendingCheckpoint被全部task接受时，就会转变为CompletedCheckpoint
            final CompletableFuture<PendingCheckpoint>
pendingCheckpointCompletableFuture =
                    checkpointPlanFuture
                            // 第一步的意思是，获取检查点的存储位置，例如文件存储、jvm
存储
                            .thenApplyAsync(
                                    plan -> {
                                        try {
                                            CheckpointIdAndStorageLocation
```

```
                                                        checkpointIdAndStorageLocation
=
                                                                // 生成checkpoint
id(AtomicLong.getAndIncrement), zk做HA的话略有区别
                                                                // 生成检查点存储路径,
一般存在FsStorage和MemoryStory
                                                                initializeCheckpoint(
                                                                        request.props,

request.externalSavepointLocation);
                                                    return new Tuple2<>(
                                                            plan,
checkpointIdAndStorageLocation);
                                                } catch (Throwable e) {
                                                    throw new CompletionException(e);
                                                }
                                            },
                                            executor)
                                    // 第二步是创建一个PendingCheckpoint
                                    .thenApplyAsync(
                                            (checkpointInfo) ->
                                                    createPendingCheckpoint(
                                                            timestamp,
                                                            request.props,
                                                            checkpointInfo.f0,
                                                            request.isPeriodic,

checkpointInfo.f1.checkpointId,

checkpointInfo.f1.checkpointStorageLocation,

request.getOnCompletionFuture()),
                                            timer);

            // ================== 总之看下最后一步 ==================

            ...

            // 执行了 triggerCheckpointRequest(CheckpointTriggerRequest request,
long timestamp, PendingCheckpoint checkpoint)
            // triggerCheckpointRequest中执行了triggerTasks()

            ...
        } catch (Throwable throwable) {
            onTriggerFailure(request, throwable);
        }
    }

    // 简而言之，就是向全部source task发送barrier
    private CompletableFuture<Void> triggerTasks(
            CheckpointTriggerRequest request, long timestamp, PendingCheckpoint
checkpoint) {
        // no exception, no discarding, everything is OK
        final long checkpointId = checkpoint.getCheckpointID();
```

```
        final CheckpointOptions checkpointOptions =
                CheckpointOptions.forConfig(
                        request.props.getCheckpointType(),

checkpoint.getCheckpointStorageLocation().getLocationReference(),
                        isExactlyOnceMode,
                        unalignedCheckpointsEnabled,
                        alignedCheckpointTimeout);

        // send messages to the tasks to trigger their checkpoints
        // 指的是全部的source function
        List<CompletableFuture<Acknowledge>> acks = new ArrayList<>();
        for (Execution execution :
checkpoint.getCheckpointPlan().getTasksToTrigger()) {
            // 当请求需要保证同步时，执行保存点任务
            if (request.props.isSynchronous()) {
                acks.add(
                        execution.triggerSynchronousSavepoint(
                                checkpointId, timestamp, checkpointOptions));
            } else {
                acks.add(execution.triggerCheckpoint(checkpointId, timestamp,
checkpointOptions));
            }
        }
        return FutureUtils.waitForAll(acks);
    }
    ...
}
```

检查点执行步骤

从上文triggerTasks()方法开始，看一下检查点的执行和barrier的流转。

从全部的source funciton开始，检查点协调器执行每个source task的Execution.triggerCheckpoint()，用于触发新的检查点。

```
// 对每个source task都执行一次
// param checkpointId: 检查点id
// param timestamp: 当前检查点被触发的时间戳
// param checkpointOptions: 部分配置
public CompletableFuture<Acknowledge> triggerCheckpoint(
            long checkpointId, long timestamp, CheckpointOptions
checkpointOptions) {
    return triggerCheckpointHelper(checkpointId, timestamp, checkpointOptions);
}
```

获取slot中的TaskManagerGateway，调用triggerCheckpoint()。

```
// param executionAttemptID: 用于标识某个task, 由Random生成
// param jobId: 当前task所属的job id
// param checkpointId: 检查点id, 由之前生成
// param timestamp: 当前检查点被触发的时间戳
// param checkpointOptions: 部分配置
public CompletableFuture<Acknowledge> triggerCheckpoint(
            ExecutionAttemptID executionAttemptID,
            JobID jobId,
            long checkpointId,
            long timestamp,
            CheckpointOptions checkpointOptions) {
    return taskExecutorGateway.triggerCheckpoint(
            executionAttemptID, checkpointId, timestamp, checkpointOptions);
    }
```

经过Gateway转发(RPC?)，检查点触发请求到达TaskExecutor和Task，最终由CheckpointableTask接口的某个实现类执行。

```
public interface CheckpointableTask {
    // This method is called to trigger a checkpoint, asynchronously by the
checkpoint coordinator.
    // 该方法被调用用来触发某个从检查点协调器异步发送过来的检查点
    // This method is called for tasks that start the checkpoints by injecting the
initial barriers, i.e., the source tasks.
    // 该方法被注入初始屏障发起检查点的task调用, 例如source task
    CompletableFuture<Boolean> triggerCheckpointAsync(
            CheckpointMetaData checkpointMetaData, CheckpointOptions
checkpointOptions);

    // This method is called when a checkpoint is triggered as a result of
receiving checkpoint barriers on all input streams.
    // 当全部的输入流接收到检查点屏障时, 调用当前方法执行检查点
    void triggerCheckpointOnBarrier(
            CheckpointMetaData checkpointMetaData,
            CheckpointOptions checkpointOptions,
            CheckpointMetricsBuilder checkpointMetrics)
            throws IOException;
}
```

举例：SourceStreamTask

```
public class SourceStreamTask {
    @Override
    public CompletableFuture<Boolean> triggerCheckpointAsync(
            CheckpointMetaData checkpointMetaData, CheckpointOptions
checkpointOptions) {
        // 这个标志位代表一个状态：实现ExternallyInducedSource接口的source不会在检查点
协调器发送请求时执行检查点，用处不明
```

```
        // 在这个标志位为1时，只有数据携带屏障才会触发检查点
        // 正常使用的话，该标志位为false，且非对齐检查点是开启该配置的
        if (!externallyInducedCheckpoints) {
            // 某种保存点规则，不考虑
            // 指的是PostCheckpointAction.TERMINATE，即
CheckpointType.SAVEPOINT_SUSPEND或SAVEPOINT_TERMINATE
            if (checkpointOptions.getCheckpointType().shouldDrain()) {
                return triggerStopWithSavepointWithDrainAsync(
                        checkpointMetaData, checkpointOptions);
            } else {
                // 这里执行的具体方法见下文
                // 该方法实现了检查点下发和恢复
                return super.triggerCheckpointAsync(checkpointMetaData,
checkpointOptions);
            }
        } else {
            // we do not trigger checkpoints here, we simply state whether we can
trigger them
            synchronized (lock) {
                return CompletableFuture.completedFuture(isRunning());
            }
        }
    }
}
```

下发检查点barrier步骤：

```
class SubtaskCheckpointCoordinatorImpl implements SubtaskCheckpointCoordinator {
    // The OperatorChain contains all operators that are executed as one chain
within a single StreamTask.
    // 这里的OperatorChain指的是不同的算子链，不同的task并发执行
    @Override
    public void checkpointState(
            CheckpointMetaData metadata,
            CheckpointOptions options,
            CheckpointMetricsBuilder metrics,
            OperatorChain<?, ?> operatorChain,
            boolean isOperatorsFinished,
            Supplier<Boolean> isRunning)
            throws Exception {

        // 检查非空：options、metrics

        // All of the following steps happen as an atomic step from the
perspective of barriers and records/watermarks/timers/callbacks.

        // 检查上一个检查点id应当小于当前检查点id，如果不是，丢弃
        // lastCheckpointId >= metadata.getCheckpointId()

        // step 0:
        // 检查该检查点是否被丢弃，检查方法是从一个HashMap中remove检查点id
        // Set<Long> abortedCheckpointIds 用来存储从job master中发送来的废弃检查点id
```

```
        lastCheckpointId = metadata.getCheckpointId();
        if (checkAndClearAbortedStatus(metadata.getCheckpointId())) {
            // broadcast cancel checkpoint marker to avoid downstream back-
pressure due to checkpoint barrier align.
            // 当remove返回true时，直接广播该请求
            operatorChain.broadcastEvent(new
CancelCheckpointMarker(metadata.getCheckpointId()));
            LOG.info(
                    "Checkpoint {} has been notified as aborted, would not trigger
any checkpoint.",
                    metadata.getCheckpointId());
            return;
        }

        // if checkpoint has been previously unaligned, but was forced to be
aligned (pointwise
        // connection), revert it here so that it can jump over output data
        // 省略一个检查点强制对齐检查，一般用不到
        ..

        // step 1:
        // 准备执行检查点，允许各个算子执行检查点前的某些操作，尽量轻量
        // 即StreamOperator.prepareSnapshotPreBarrier(long checkpointId)
        // 比如sink算子，可能会提交缓冲的数据
        // 该方法不应该实现状态的持久化，仅仅是用于检查点屏障之前发出一些数据。
        // 一般来说，除了sink算子这里什么都不做
        operatorChain.prepareSnapshotPreBarrier(metadata.getCheckpointId());

        // step 2:
        // 向下游发送检查点屏障，通过RecordWriter
        operatorChain.broadcastEvent(
                new CheckpointBarrier(metadata.getCheckpointId(),
metadata.getTimestamp(), options),
                options.isUnalignedCheckpoint());

        // step 3:
        // 当检查点时非对齐时，需要对in-flight数据做操作
        // 正常情况下是不存在非对齐检查点的
        if (options.isUnalignedCheckpoint()) {
            // output data already written while broadcasting event
            channelStateWriter.finishOutput(metadata.getCheckpointId());
        }

        // step 4:
        // 执行状态快照，异步执行
        Map<OperatorID, OperatorSnapshotFutures> snapshotFutures =
                new HashMap<>(operatorChain.getNumberOfOperators());

        try {
            // if中takeSnapshotSync方法会填充snapshotFutures的key value，value指的是
算子中的状态存储结果OperatorSnapshotFutures
            // 填充方式是状态后端执行state handler，调用快照策略
            // OperatorSnapshotFutures记录了key state、operator state的托管状态和原
始状态、input output channel信息
```

```
            // 具体的状态序列化过程见：状态详解.md
            // 填充过程中会执行CheckpointedFunction接口的snapshotState，如果你的UDF算
子实现了该接口。
            // 执行完UDF的CheckpointedFunction，才会执行operator state和keyed state
的持久化
            // 顺便一提，Kafka source的检查点执行逻辑在
KafkaSourceReader#snapshotState，和检查点一起提交offsets也在这里
            if (takeSnapshotSync(
                    snapshotFutures, metadata, metrics, options, operatorChain,
isRunning)) {
                // 其中实现了异步的完成与提交
                // 其中实现了检查点的存储和通过CheckpointResponder(Gateway)上报
                finishAndReportAsync(
                        snapshotFutures,
                        metadata,
                        metrics,
                        operatorChain.isFinishedOnRestore(),
                        isOperatorsFinished,
                        isRunning);
            } else {
                cleanup(snapshotFutures, metadata, metrics, new
Exception("Checkpoint declined"));
            }
        } catch (Exception ex) {
            // 释放资源
            cleanup(snapshotFutures, metadata, metrics, ex);
            throw ex;
        }
    }
}
```

下游算子接受检查点屏障

如上文所示，operatorChain.broadcastEvent将检查点屏障广播到下游的全部分区。 随后在下游的数据流input中，存在如下判断：

```
    // 详见AbstractStreamTaskNetworkInput:[110]
    Optional<BufferOrEvent> bufferOrEvent = checkpointedInputGate.pollNext();
    if (bufferOrEvent.isPresent()) {
        if (bufferOrEvent.get().isBuffer()) {
            processBuffer(bufferOrEvent.get());
        } else {
            // 如果接收到的是检查点屏障，则进入改方法
            return processEvent(bufferOrEvent.get());
        }
    } else {
        if (checkpointedInputGate.isFinished()) {
            checkState(
                    checkpointedInputGate.getAvailableFuture().isDone(),
                    "Finished BarrierHandler should be available");
            return DataInputStatus.END_OF_INPUT;
```

```
            }
            return DataInputStatus.NOTHING_AVAILABLE;
        }
```

检查点模式 精确一次和至少一次

检测点模式定义了当job发生故障时，程序能提供的一致性保证。 简单点的描述就是从{精准但是慢，快但是不精准}中选择一个模式。

```
public enum CheckpointingMode {
    // 慢但是精准，在某次检查点到来之时，是什么状态就是什么状态。
    // 实现起来就是保证一个算子的全部输入流检查点对齐，对于数据倾斜严重的算子来说，会具有延迟甚至导致超时失败。
    EXACTLY_ONCE,
    // 快但是不精准，某次检查点可能会导致错误的状态。
    // 实现起来就是不保证对齐。
    AT_LEAST_ONCE
}
```

简单了解一下两者的逻辑：

- EXACTLY_ONCE在某个input channel接受到某个barrier时，标记对齐的数据的开始和结束，将其加入阻塞队列，等待全部channel完成后，触发算子检查点，完成后上报。
- EXACTLY_ONCE的详细实现可以看 org.apache.flink.streaming.runtime.io.checkpointing.SingleCheckpointBarrierHandler
- AT_LEAST_ONCE在某个input channel接受到某个barrier时，直接执行检查点，等待全部channel接受并完成后完成后上报。
- AT_LEAST_ONCE的详细实现可以看 org.apache.flink.streaming.runtime.io.checkpointing.CheckpointBarrierTracker

通常设置检查点模式的方式为：

```
    // DEFAULT_MODE 即是 CheckpointingMode.EXACTLY_ONCE
    env.getCheckpointConfig().setCheckpointingMode(CheckpointConfig.DEFAULT_MODE);
```

检查点模式的运作方式，详见InputProcessorUtil#createCheckpointBarrierHandler，一个BarrierHandler的作用是从input channel中接受barrier并调用快照策略，如下：

```
public class InputProcessorUtil {
    // 在StreamTask中被初始化，例如OneInputStreamTask#init
    public static CheckpointBarrierHandler createCheckpointBarrierHandler(
            CheckpointableTask toNotifyOnCheckpoint,
            StreamConfig config,
            SubtaskCheckpointCoordinator checkpointCoordinator,
```

```
            String taskName,
            List<IndexedInputGate>[] inputGates,
            List<StreamTaskSourceInput<?>> sourceInputs,
            MailboxExecutor mailboxExecutor,
            TimerService timerService) {

        CheckpointableInput[] inputs =
                Stream.<CheckpointableInput>concat(

Arrays.stream(inputGates).flatMap(Collection::stream),
                                sourceInputs.stream())

.sorted(Comparator.comparing(CheckpointableInput::getInputGateIndex))
                        .toArray(CheckpointableInput[]::new);

        Clock clock = SystemClock.getInstance();
        switch (config.getCheckpointMode()) {
            case EXACTLY_ONCE:
                int numberOfChannels =
                        (int)
                                Arrays.stream(inputs)
                                        .mapToLong(gate ->
gate.getChannelInfos().size())
                                        .sum();
                return createBarrierHandler(
                        toNotifyOnCheckpoint,
                        config,
                        checkpointCoordinator,
                        taskName,
                        mailboxExecutor,
                        timerService,
                        inputs,
                        clock,
                        numberOfChannels);
            case AT_LEAST_ONCE:
                // 如下，非对齐检查点禁止使用AT_LEAST_ONCE
                if (config.isUnalignedCheckpointsEnabled()) {
                    throw new IllegalStateException(
                            "Cannot use unaligned checkpoints with AT_LEAST_ONCE "
                                    + "checkpointing mode");
                }
                int numInputChannels =
                        Arrays.stream(inputs)

.mapToInt(CheckpointableInput::getNumberOfInputChannels)
                                .sum();
                return new CheckpointBarrierTracker(
                        numInputChannels,
                        toNotifyOnCheckpoint,
                        clock,
                        config.getConfiguration()
                                .get(
                                        ExecutionCheckpointingOptions
```

```
.ENABLE_CHECKPOINTS_AFTER_TASKS_FINISH));
            default:
                throw new UnsupportedOperationException(
                        "Unrecognized Checkpointing Mode: " +
config.getCheckpointMode());
        }
    }
}
```

两者的代码上区别就不放了，大致为EXACTLY_ONCE在接受到barrier时，会执行如下方法：

```
final class ChannelState {
    public void blockChannel(InputChannelInfo channelInfo) {
        inputs[channelInfo.getGateIdx()].blockConsumption(channelInfo);
        blockedChannels.add(channelInfo);
    }

    public void unblockAllChannels() throws IOException {
        for (InputChannelInfo blockedChannel : blockedChannels) {
            inputs[blockedChannel.getGateIdx()].resumeConsumption(blockedChannel);
        }
        blockedChannels.clear();
    }
}

public interface CheckpointableInput {
    void blockConsumption(InputChannelInfo channelInfo);

    void resumeConsumption(InputChannelInfo channelInfo) throws IOException;
}
```

注意，block和unblock都是没有debug日志的。

通过debug日志了解检查点流程

10.182.63.222:/root/dafeihou/flink/taskmanagerlog.txt 和 jobmanagerlog.txt，记录了检查点相关的debug日志。 摘取了其中某个算子的检查点相关日志，见附件。 总体流程如下：

1. 由CheckpointCoordinator生成PendingCheckpoint，{checkpointID}，{checkpointType}，{timestamp}，{job}

   > org.apache.flink.runtime.checkpoint.CheckpointCoordinator:[790] - Triggering checkpoint 1 (type=CHECKPOINT) @ 1664435313052 for job f5a0ed861cf340f8a467212deaabff1e.

2. 通过SourceCoordinator向全部的source算子发送消息

   > 2022-09-29 07:08:33,087 DEBUG [,] [SourceCoordinator-Source: operator source:read white list from kafka] org.apache.flink.runtime.source.coordinator.SourceCoordinator:[240] - Taking a state snapshot on operator Source: operator source:read white list from kafka for checkpoint 1

3. TaskExecutor接收到检查点请求，{checkpointId}，{checkpointTimestamp}，{executionAttemptID}，次数为source task的个数

> org.apache.flink.runtime.taskexecutor.TaskExecutor:[940] - Trigger checkpoint 1@1664435313052 for 42f8329b9554903bf03e41100b79fde0.

4. StreamTask接收到TaskExecutor传入的请求，{getCheckpointId}，{checkpointType}，{task name}，次数为source task的个数

> org.apache.flink.streaming.runtime.tasks.StreamTask:[1300] - Starting checkpoint 1 CHECKPOINT on task Source: operator source:read white list from kafka (1/1)#0

5. SingleCheckpointBarrierHandler接收到广播的barrier，每个input channel都会出现，{task name}，{channel info}，{barrierId}

> org.apache.flink.streaming.runtime.io.checkpointing.SingleCheckpointBarrierHandler:[218] - operator : config white list operator (4/4)#0 (956e5f18e295ba34553049ba9969bc98): Received barrier from channel InputChannelInfo{gateIdx=1, inputChannelIdx=0} @ 1.

6. 与此同时，非source的算子也开始接受到barrier，即非source算子的StreamTask开始步骤3。
7. 快照策略执行后，打印一行SnapshotStrategyRunner日志，过长就不放了。
8. 异步上报

> 2022-09-29 07:08:33,344 DEBUG [,] [operator : window rule (4/4)#0] org.apache.flink.streaming.runtime.tasks.StreamTask:[1418] - Notify checkpoint 1 complete on task operator : window rule (4/4)#0

9. JobManager依次接收到检查点完成的信息

> 2022-09-29 07:08:33,155 DEBUG [,] [jobmanager-io-thread-1] org.apache.flink.runtime.checkpoint.CheckpointCoordinator:[1078] - Received acknowledge message for checkpoint 1 from task 863ad4f0880c75e94ec6799e24c139e0 of job f5a0ed861cf340f8a467212deaabff1e at 10.244.0.37:46055-55ebb3 @ 10.244.0.37 (dataPort=46871).

10. JobManager确定全部task完成检查点

> 2022-09-29 07:08:33,305 INFO [,] [jobmanager-io-thread-2] org.apache.flink.runtime.checkpoint.CheckpointCoordinator:[1260] - Completed checkpoint 1 for job f5a0ed861cf340f8a467212deaabff1e (217867 bytes, checkpointDuration=247 ms, finalizationTime=6 ms).

11. 打印接收到的全部算子检查点情况，过长不放
12. 通知source task已完成检查点

> 2022-09-29 07:08:33,312 INFO [,] [SourceCoordinator-Source: operator source:read config white list from kafka] org.apache.flink.runtime.source.coordinator.SourceCoordinator:[265] - Marking checkpoint 1 as completed for source Source: operator source:read config white list from kafka.

13. 完成，后续就触发新的检查点

确认检查点完成

上文中提到，当算子完成检查点时，会通知调度器；当某个检查点需要的全部算子均返回SUCCESS结果后，对应的PendingCheckpoint置换为completeCheckpoint。

执行具体代码详见CheckpointCoordinator.receiveAcknowledgeMessage()。

交换检查点信息

检查点恢复过程

Flink job发生异常自动恢复或者保存点恢复时，经过检查点协调器调用restoreLatestCheckpointedStateInternal方法，会打印如下日志

```
2022-10-04 11:06:16.383 [flink-akka.actor.default-dispatcher-6] DEBUG
o.a.flink.runtime.checkpoint.CheckpointCoordinator -Status of the shared state
registry of job c631171b2148afba047fa0b582dd1d4d after restore:
SharedStateRegistry{registeredStates={}}.
```

紧接着从上一次执行成功的路径下获取检查点数据，具体的代码不放，详见CheckpointCoordinator#restoreLatestCheckpointedStateInternal。

如果用户开启检查点并且存在上一次完成的检查点，则打印如下日志：

```
2022-10-04 11:06:16.383 [flink-akka.actor.default-dispatcher-6] INFO
o.a.flink.runtime.checkpoint.CheckpointCoordinator -Restoring job
c631171b2148afba047fa0b582dd1d4d from Checkpoint 4 @ 1664852766309 for
c631171b2148afba047fa0b582dd1d4d located at
file:/E:/checkpoint/c631171b2148afba047fa0b582dd1d4d/chk-4.
``

如果不存在上一次完成的检查点，则打印：No checkpoint found during restore.

因为失败重启不存在算子增删改或者并行度缩放问题，所以校验不严格，和savepoint不一样。

## 检查点存储

在程序中指定检查点的存储：

```java
env.getCheckpointConfig().setCheckpointStorage(CheckpointStorage);
// or
// 这样的话相当于直接指定了使用FileSystemCheckpointStorage
env.getCheckpointConfig().setCheckpointStorage(String or URI or Path);
```

```
// CheckpointStorage定义了状态后端如何存储和实现容错，例如JobManagerCheckpointStorage
和FileSystemCheckpointStorage
// JobManagerCheckpointStorage在JobManager的内存中存储状态信息，这使得存储很轻量级并且
不存在外部依赖
// 但是存储在内存中易丢失且只能支持大小较小的状态信息。
```
```

```
// FileSystemCheckpointStorage使用文件系统存储，例如HDFS，NFS Drives，S3，and GCS，
这些存储支持较大的状态信息。
// FileSystemCheckpointStorage针对大部分应用来说是比较合适的。
// 实现字节存储和恢复，JobManager通过字节存储检查点和恢复元数据，并且也存储keyed和
operator状态
// 实现类需要实现序列化接口 Serializable
// 线程安全，因为多个并行流可能同时创建
public interface CheckpointStorage {
    // 将给定的、指向检查点/保存点的指针解析为检查点位置；该位置支持读取检查点数据或处置
检查点存储位置
    CompletedCheckpointStorageLocation resolveCheckpoint(String externalPointer)
throws IOException;

    // 为给定job创建存储
    // 在检查点协调器的构造方法中，调用了该方法，详见CheckpointCoordinator:[325]
    // 在该方法后，又调用了checkpointStorageView.initializeBaseLocations()，创建了文
件目录
    // 实际上在创建目录的时候，会自动加上chk-{checkpoint id}，详见
AbstractFsCheckpointStorageAccess:[223]
    CheckpointStorageAccess createCheckpointStorage(JobID jobId) throws
IOException;
}

public interface CompletedCheckpointStorageLocation {
    // 获取指向检查点的外部指针
    String getExternalPointer();
    // 获取检查点元数据的状态句柄
    StreamStateHandle getMetadataHandle();
    // 处置存储位置，通常用于处理检查点存储的基本结构
    void disposeStorageLocation() throws IOException;
}

// 该接口实现两个方法：
// 1. CheckpointStorageCoordinatorView接口中定义的管理员方法：创建检查点或保存点路径
// 2. CheckpointStorageWorkerView接口中定义的方法实现了检查点数据和元数据的持久化存储，
存储路径为CheckpointStorageCoordinatorView中创建的路径。
public interface CheckpointStorageAccess
        extends CheckpointStorageCoordinatorView, CheckpointStorageWorkerView {}
```

这里以FileSystemCheckpointStorage为例，看一看具体的处理流程。

首先是实现CheckpointStorage的FileSystemCheckpointStorage，将检查点状态生成文件并存储在文件系统中。

使用vvp默认的检查点存储为：*state.backend: filesystem*，默认的checkpoint目录为：*/vvp/flink-jobs/namespaces/default/jobs/{id}/checkpoints/{id}/chk-{count}*，修改目录的方式为：*state.checkpoints.dir: 'file:///var/local/checkpoint'* 或 *'hdfs://namenode:port/var/local/checkpoint'*。

```
public class FileSystemCheckpointStorage {
    // 两个path，分别为检查点地址和保存点地址
    private final ExternalizedSnapshotLocation location;
    ...
```

```
    public FileSystemCheckpointStorage(String checkpointDirectory) {...}
    public FileSystemCheckpointStorage(Path checkpointDirectory) {...}
    public FileSystemCheckpointStorage(URI checkpointDirectory) {...}

    /*
     *
org.apache.flink.runtime.state.storage.FileSystemCheckpointStorage#createFromConfi
g会读取config中的state.checkpoints.dir
     * 详见FileSystemCheckpointStorage:[293]
    */

    @Override
    public CompletedCheckpointStorageLocation resolveCheckpoint(String pointer)
throws IOException {
        return
AbstractFsCheckpointStorageAccess.resolveCheckpointPointer(pointer);
    }

    @Override
    public CheckpointStorageAccess createCheckpointStorage(JobID jobId) throws
IOException {
        checkNotNull(jobId, "jobId");
        return new FsCheckpointStorageAccess(
                location.getBaseCheckpointPath(),
                location.getBaseSavepointPath(),
                jobId,
                getMinFileSizeThreshold(),
                getWriteBufferSize());
    }
}
```

附件

某个算子的某次检查点日志

```
2022-09-29 07:08:20,769 DEBUG [,]
org.apache.flink.streaming.runtime.tasks.StreamTask:[81] - #The configuration
state.checkpoint-storage has not be set in the current sessions flink-conf.yaml.
Falling back to a default CheckpointStorage type. Users are strongly encouraged
explicitly set this configuration so they understand how their applications are
checkpointing snapshots for fault-tolerance.
2022-09-29 07:08:20,770 INFO  [,]
org.apache.flink.streaming.runtime.tasks.StreamTask:[263] - #Checkpoint storage is
set to 'filesystem': (checkpoints "file:/var/local/checkpoint")


因为上一个算子拥有并行度4，即4个output channel，所以当前算子的每个subtask都拥有4个input
channel。
2022-09-29 07:08:33,119 DEBUG [,]
org.apache.flink.streaming.runtime.io.checkpointing.SingleCheckpointBarrierHandler
```

```
:[218] - #operator : window rule (4/4)#0 (5b3167f121239ea3db5de1d9d5773b1e):
Received barrier from channel InputChannelInfo{gateIdx=0, inputChannelIdx=1} @ 1.
2022-09-29 07:08:33,121 DEBUG [,]
org.apache.flink.streaming.runtime.io.checkpointing.SingleCheckpointBarrierHandler
:[218] - #operator : window rule (4/4)#0 (5b3167f121239ea3db5de1d9d5773b1e):
Received barrier from channel InputChannelInfo{gateIdx=0, inputChannelIdx=3} @ 1.
2022-09-29 07:08:33,121 DEBUG [,]
org.apache.flink.streaming.runtime.io.checkpointing.SingleCheckpointBarrierHandler
:[218] - #operator : window rule (4/4)#0 (5b3167f121239ea3db5de1d9d5773b1e):
Received barrier from channel InputChannelInfo{gateIdx=0, inputChannelIdx=0} @ 1.
2022-09-29 07:08:33,125 DEBUG [,]
org.apache.flink.streaming.runtime.io.checkpointing.SingleCheckpointBarrierHandler
:[218] - #operator : window rule (4/4)#0 (5b3167f121239ea3db5de1d9d5773b1e):
Received barrier from channel InputChannelInfo{gateIdx=0, inputChannelIdx=2} @ 1.
```

当某个算子的全部input channel都接收到barrier时，打印该日志。具体见AbstractAlignedBarrierHandlerState#barrierReceived:[66]，一个subtask只会打印一次。

```
2022-09-29 07:08:33,126 DEBUG [,]
org.apache.flink.streaming.runtime.io.checkpointing.SingleCheckpointBarrierHandler
:[282] - #operator : window rule (4/4)#0 (5b3167f121239ea3db5de1d9d5773b1e):
Triggering checkpoint 1 on the barrier announcement at 1664435313052.
```

在上一行打完日志后在同一个方法内被调用。该方法当仅当算子全部input都接收到barrier时被调用用作触发检查点的结果。详见CheckpointableTask#triggerCheckpointOnBarrier。

```
2022-09-29 07:08:33,127 DEBUG [,]
org.apache.flink.streaming.runtime.tasks.StreamTask:[1300] - #Starting checkpoint
1 CHECKPOINT on task operator : window rule (4/4)#0
```

在StreamTask确认全部input接收到barrier时，通过subtaskCheckpointCoordinator#checkpointState和operatorChain#snapshotState通知各个算子执行检查点。
各个算子通过stateHandler对状态做操作。
日志模板是：{} ({}, synchronous part) in thread {} took {} ms.
因为状态分为operator state和keyed state，所以需要打印两次日志，且不同状态的状态后端不同，例如这次operator state使用DefaultOperatorStateBackend，keyed state使用Heap backend。
关于两种state的执行逻辑详见StreamOperatorStateHandler#snapshotState，即StreamOperatorStateHandler:[186]

```
2022-09-29 07:08:33,128 DEBUG [,]
org.apache.flink.runtime.state.SnapshotStrategyRunner:[120] -
#DefaultOperatorStateBackend snapshot (FsCheckpointStorageLocation
{fileSystem=org.apache.flink.core.fs.SafetyNetWrapperFileSystem@2e5c1005,
checkpointDirectory=file:/var/local/checkpoint/f5a0ed861cf340f8a467212deaabff1e/ch
k-1,
sharedStateDirectory=file:/var/local/checkpoint/f5a0ed861cf340f8a467212deaabff1e/s
hared,
taskOwnedStateDirectory=file:/var/local/checkpoint/f5a0ed861cf340f8a467212deaabff1
e/taskowned,
metadataFilePath=file:/var/local/checkpoint/f5a0ed861cf340f8a467212deaabff1e/chk-
1/_metadata, reference=(default), fileStateSizeThreshold=20480,
writeBufferSize=20480}, synchronous part) in thread Thread[operator : window rule
```

```
(4/4)#0,5,Flink Task Threads] took 0 ms.
2022-09-29 07:08:33,217 DEBUG [,]
org.apache.flink.runtime.state.SnapshotStrategyRunner:[120] - #Heap backend
snapshot (FsCheckpointStorageLocation
{fileSystem=org.apache.flink.core.fs.SafetyNetWrapperFileSystem@2e5c1005,
checkpointDirectory=file:/var/local/checkpoint/f5a0ed861cf340f8a467212deaabff1e/ch
k-1,
sharedStateDirectory=file:/var/local/checkpoint/f5a0ed861cf340f8a467212deaabff1e/s
hared,
taskOwnedStateDirectory=file:/var/local/checkpoint/f5a0ed861cf340f8a467212deaabff1
e/taskowned,
metadataFilePath=file:/var/local/checkpoint/f5a0ed861cf340f8a467212deaabff1e/chk-
1/_metadata, reference=(default), fileStateSizeThreshold=20480,
writeBufferSize=20480}, synchronous part) in thread Thread[operator : window rule
(4/4)#0,5,Flink Task Threads] took 87 ms.


上文状态后端执行快照后，打印日志
2022-09-29 07:08:33,218 DEBUG [,]
org.apache.flink.streaming.runtime.tasks.SubtaskCheckpointCoordinatorImpl:[617] -
#operator : window rule (4/4)#0 - finished synchronous part of checkpoint 1.
Alignment duration: 6 ms, snapshot duration -1 ms, is unaligned checkpoint : false
2022-09-29 07:08:33,219 DEBUG [,]
org.apache.flink.streaming.runtime.io.checkpointing.SingleCheckpointBarrierHandler
:[272] - #operator : window rule (4/4)#0 (5b3167f121239ea3db5de1d9d5773b1e): All
the channels are aligned for checkpoint 1.


这里已经是异步执行检查点上报了
2022-09-29 07:08:33,219 DEBUG [,] [AsyncOperations-thread-1]
org.apache.flink.streaming.runtime.tasks.AsyncCheckpointRunnable:[109] - #operator
: window rule (4/4)#0 - started executing asynchronous part of checkpoint 1.
Asynchronous start delay: 1 ms
2022-09-29 07:08:33,236 DEBUG [,] [AsyncOperations-thread-1]
org.apache.flink.streaming.runtime.tasks.AsyncCheckpointRunnable:[232] - #operator
: window rule (4/4)#0 - finished asynchronous part of checkpoint 1. Asynchronous
duration: 15 ms
2022-09-29 07:08:33,344 DEBUG [,]
org.apache.flink.streaming.runtime.tasks.StreamTask:[1418] - #Notify checkpoint 1
complete on task operator : window rule (4/4)#0
2022-09-29 07:08:33,344 DEBUG [,]
org.apache.flink.streaming.runtime.tasks.SubtaskCheckpointCoordinatorImpl:[344] -
#Notification of completed checkpoint 1 for task operator : window rule (4/4)#0
```

jobmanager日志

```
2022-09-29 07:08:33,080 INFO  [,] [Checkpoint Timer]
org.apache.flink.runtime.checkpoint.CheckpointCoordinator:[790] - Triggering
checkpoint 1 (type=CHECKPOINT) @ 1664435313052 for job
f5a0ed861cf340f8a467212deaabff1e.
2022-09-29 07:08:33,087 DEBUG [,] [SourceCoordinator-Source: operator source:read
```

```
threat log from kafka -> Sink: sink : sink to etl]
org.apache.flink.runtime.source.coordinator.SourceCoordinator:[240] - Taking a
state snapshot on operator Source: operator source:read threat log from kafka ->
Sink: sink : sink to etl for checkpoint 1
2022-09-29 07:08:33,087 DEBUG [,] [SourceCoordinator-Source: operator source:read
config white list from kafka]
org.apache.flink.runtime.source.coordinator.SourceCoordinator:[240] - Taking a
state snapshot on operator Source: operator source:read config white list from
kafka for checkpoint 1
2022-09-29 07:08:33,087 DEBUG [,] [SourceCoordinator-Source: operator source:read
white list from kafka]
org.apache.flink.runtime.source.coordinator.SourceCoordinator:[240] - Taking a
state snapshot on operator Source: operator source:read white list from kafka for
checkpoint 1
2022-09-29 07:08:33,155 DEBUG [,] [jobmanager-io-thread-1]
org.apache.flink.runtime.checkpoint.CheckpointCoordinator:[1078] - Received
acknowledge message for checkpoint 1 from task 863ad4f0880c75e94ec6799e24c139e0 of
job f5a0ed861cf340f8a467212deaabff1e at 10.244.0.37:46055-55ebb3 @ 10.244.0.37
(dataPort=46871).
2022-09-29 07:08:33,156 DEBUG [,] [jobmanager-io-thread-2]
org.apache.flink.runtime.checkpoint.CheckpointCoordinator:[1078] - Received
acknowledge message for checkpoint 1 from task 6ef893c3eca29ad82bfff3097bb7d7bf of
job f5a0ed861cf340f8a467212deaabff1e at 10.244.0.37:46055-55ebb3 @ 10.244.0.37
(dataPort=46871).
2022-09-29 07:08:33,156 DEBUG [,] [jobmanager-io-thread-2]
org.apache.flink.runtime.checkpoint.CheckpointCoordinator:[1078] - Received
acknowledge message for checkpoint 1 from task ea6bdf15c8df4784b513510339464e81 of
job f5a0ed861cf340f8a467212deaabff1e at 10.244.0.37:46055-55ebb3 @ 10.244.0.37
(dataPort=46871).
2022-09-29 07:08:33,156 DEBUG [,] [jobmanager-io-thread-1]
org.apache.flink.runtime.checkpoint.CheckpointCoordinator:[1078] - Received
acknowledge message for checkpoint 1 from task 8c2660f8d46cca660e75f6d8d581447c of
job f5a0ed861cf340f8a467212deaabff1e at 10.244.0.37:46055-55ebb3 @ 10.244.0.37
(dataPort=46871).
2022-09-29 07:08:33,156 DEBUG [,] [jobmanager-io-thread-2]
org.apache.flink.runtime.checkpoint.CheckpointCoordinator:[1078] - Received
acknowledge message for checkpoint 1 from task 0a2b09ce2fd89b8f3188a944b289e091 of
job f5a0ed861cf340f8a467212deaabff1e at 10.244.0.37:46055-55ebb3 @ 10.244.0.37
(dataPort=46871).
2022-09-29 07:08:33,157 DEBUG [,] [jobmanager-io-thread-1]
org.apache.flink.runtime.checkpoint.CheckpointCoordinator:[1078] - Received
acknowledge message for checkpoint 1 from task f757b498b79a610b8e70c85abdea8466 of
job f5a0ed861cf340f8a467212deaabff1e at 10.244.0.37:46055-55ebb3 @ 10.244.0.37
(dataPort=46871).
2022-09-29 07:08:33,245 DEBUG [,] [jobmanager-io-thread-2]
org.apache.flink.runtime.checkpoint.CheckpointCoordinator:[1078] - Received
acknowledge message for checkpoint 1 from task abb2903eaa201ce35a1cda9743421823 of
job f5a0ed861cf340f8a467212deaabff1e at 10.244.0.37:46055-55ebb3 @ 10.244.0.37
(dataPort=46871).
2022-09-29 07:08:33,246 DEBUG [,] [jobmanager-io-thread-1]
org.apache.flink.runtime.checkpoint.CheckpointCoordinator:[1078] - Received
acknowledge message for checkpoint 1 from task 11c5dd9370b5ed90aeaf22c3008a8228 of
job f5a0ed861cf340f8a467212deaabff1e at 10.244.0.37:46055-55ebb3 @ 10.244.0.37
(dataPort=46871).
```

```
2022-09-29 07:08:33,248 DEBUG [,] [jobmanager-io-thread-2]
org.apache.flink.runtime.checkpoint.CheckpointCoordinator:[1078] - Received
acknowledge message for checkpoint 1 from task c9a9a0da896203a33536ce459c538e43 of
job f5a0ed861cf340f8a467212deaabff1e at 10.244.0.37:46055-55ebb3 @ 10.244.0.37
(dataPort=46871).
2022-09-29 07:08:33,248 DEBUG [,] [jobmanager-io-thread-1]
org.apache.flink.runtime.checkpoint.CheckpointCoordinator:[1078] - Received
acknowledge message for checkpoint 1 from task 2592af55485ee4588eac5e7653a1798e of
job f5a0ed861cf340f8a467212deaabff1e at 10.244.0.37:46055-55ebb3 @ 10.244.0.37
(dataPort=46871).
2022-09-29 07:08:33,249 DEBUG [,] [jobmanager-io-thread-2]
org.apache.flink.runtime.checkpoint.CheckpointCoordinator:[1078] - Received
acknowledge message for checkpoint 1 from task c35ba8b594edde02e3d82b34101e4534 of
job f5a0ed861cf340f8a467212deaabff1e at 10.244.0.37:46055-55ebb3 @ 10.244.0.37
(dataPort=46871).
2022-09-29 07:08:33,250 DEBUG [,] [jobmanager-io-thread-1]
org.apache.flink.runtime.checkpoint.CheckpointCoordinator:[1078] - Received
acknowledge message for checkpoint 1 from task 956e5f18e295ba34553049ba9969bc98 of
job f5a0ed861cf340f8a467212deaabff1e at 10.244.0.37:46055-55ebb3 @ 10.244.0.37
(dataPort=46871).
2022-09-29 07:08:33,252 DEBUG [,] [jobmanager-io-thread-2]
org.apache.flink.runtime.checkpoint.CheckpointCoordinator:[1078] - Received
acknowledge message for checkpoint 1 from task 88576b314be5942cebd63072095276d2 of
job f5a0ed861cf340f8a467212deaabff1e at 10.244.0.37:46055-55ebb3 @ 10.244.0.37
(dataPort=46871).
2022-09-29 07:08:33,253 DEBUG [,] [jobmanager-io-thread-1]
org.apache.flink.runtime.checkpoint.CheckpointCoordinator:[1078] - Received
acknowledge message for checkpoint 1 from task e455a67aa8f54d84fe0d6af0986fe72b of
job f5a0ed861cf340f8a467212deaabff1e at 10.244.0.37:46055-55ebb3 @ 10.244.0.37
(dataPort=46871).
2022-09-29 07:08:33,254 DEBUG [,] [jobmanager-io-thread-2]
org.apache.flink.runtime.checkpoint.CheckpointCoordinator:[1078] - Received
acknowledge message for checkpoint 1 from task 2a95883e449e34a780a21f03614ac759 of
job f5a0ed861cf340f8a467212deaabff1e at 10.244.0.37:46055-55ebb3 @ 10.244.0.37
(dataPort=46871).
2022-09-29 07:08:33,257 DEBUG [,] [jobmanager-io-thread-1]
org.apache.flink.runtime.checkpoint.CheckpointCoordinator:[1078] - Received
acknowledge message for checkpoint 1 from task c78825b58df97e2b0ab5cc303bffc22a of
job f5a0ed861cf340f8a467212deaabff1e at 10.244.0.37:46055-55ebb3 @ 10.244.0.37
(dataPort=46871).
2022-09-29 07:08:33,259 DEBUG [,] [jobmanager-io-thread-2]
org.apache.flink.runtime.checkpoint.CheckpointCoordinator:[1078] - Received
acknowledge message for checkpoint 1 from task 66b61e38b73816321cf5a15f903e1544 of
job f5a0ed861cf340f8a467212deaabff1e at 10.244.0.37:46055-55ebb3 @ 10.244.0.37
(dataPort=46871).
2022-09-29 07:08:33,259 DEBUG [,] [jobmanager-io-thread-1]
org.apache.flink.runtime.checkpoint.CheckpointCoordinator:[1078] - Received
acknowledge message for checkpoint 1 from task 42f8329b9554903bf03e41100b79fde0 of
job f5a0ed861cf340f8a467212deaabff1e at 10.244.0.37:46055-55ebb3 @ 10.244.0.37
(dataPort=46871).
2022-09-29 07:08:33,261 DEBUG [,] [jobmanager-io-thread-2]
org.apache.flink.runtime.checkpoint.CheckpointCoordinator:[1078] - Received
acknowledge message for checkpoint 1 from task 5b3167f121239ea3db5de1d9d5773b1e of
job f5a0ed861cf340f8a467212deaabff1e at 10.244.0.37:46055-55ebb3 @ 10.244.0.37
```

```
(dataPort=46871).
2022-09-29 07:08:33,261 DEBUG [,] [jobmanager-io-thread-1]
org.apache.flink.runtime.checkpoint.CheckpointCoordinator:[1078] - Received
acknowledge message for checkpoint 1 from task 4c51534c087d20786909c4ebc9fef7ab of
job f5a0ed861cf340f8a467212deaabff1e at 10.244.0.37:46055-55ebb3 @ 10.244.0.37
(dataPort=46871).
2022-09-29 07:08:33,262 DEBUG [,] [jobmanager-io-thread-2]
org.apache.flink.runtime.checkpoint.CheckpointCoordinator:[1078] - Received
acknowledge message for checkpoint 1 from task 3ed079ef21b0d8e837cafc23f9886452 of
job f5a0ed861cf340f8a467212deaabff1e at 10.244.0.37:46055-55ebb3 @ 10.244.0.37
(dataPort=46871).
2022-09-29 07:08:33,274 DEBUG [,] [jobmanager-io-thread-1]
org.apache.flink.runtime.checkpoint.CheckpointCoordinator:[1078] - Received
acknowledge message for checkpoint 1 from task 363490e40fac484b86984329b80a2e36 of
job f5a0ed861cf340f8a467212deaabff1e at 10.244.0.37:46055-55ebb3 @ 10.244.0.37
(dataPort=46871).
2022-09-29 07:08:33,275 DEBUG [,] [jobmanager-io-thread-2]
org.apache.flink.runtime.checkpoint.CheckpointCoordinator:[1078] - Received
acknowledge message for checkpoint 1 from task 6bb8fcaff0de0c6aca15312516ec5c4f of
job f5a0ed861cf340f8a467212deaabff1e at 10.244.0.37:46055-55ebb3 @ 10.244.0.37
(dataPort=46871).
2022-09-29 07:08:33,276 DEBUG [,] [jobmanager-io-thread-1]
org.apache.flink.runtime.checkpoint.CheckpointCoordinator:[1078] - Received
acknowledge message for checkpoint 1 from task 13447b9c4767f2eaa970f617d8036a09 of
job f5a0ed861cf340f8a467212deaabff1e at 10.244.0.37:46055-55ebb3 @ 10.244.0.37
(dataPort=46871).
2022-09-29 07:08:33,277 DEBUG [,] [jobmanager-io-thread-2]
org.apache.flink.runtime.checkpoint.CheckpointCoordinator:[1078] - Received
acknowledge message for checkpoint 1 from task cb4e33a237a09e497cb867978a308bf8 of
job f5a0ed861cf340f8a467212deaabff1e at 10.244.0.37:46055-55ebb3 @ 10.244.0.37
(dataPort=46871).
2022-09-29 07:08:33,305 INFO  [,] [jobmanager-io-thread-2]
org.apache.flink.runtime.checkpoint.CheckpointCoordinator:[1260] - Completed
checkpoint 1 for job f5a0ed861cf340f8a467212deaabff1e (217867 bytes,
checkpointDuration=247 ms, finalizationTime=6 ms).
2022-09-29 07:08:33,306 DEBUG [,] [jobmanager-io-thread-2]
org.apache.flink.runtime.checkpoint.CheckpointCoordinator:[1278] - Checkpoint
state: OperatorState(operatorID: cbc357ccb763df2852fee8c4fc7d55f2, parallelism: 1,
maxParallelism: 128, coordinatorState: 296 bytes, sub task states: 1, total size
(bytes): 1033), OperatorState(operatorID: 4c860d0bec75b7401a18b688603dd4d0,
parallelism: 1, maxParallelism: 128, coordinatorState: (none), sub task states: 1,
total size (bytes): 0), OperatorState(operatorID:
3f35542b97b723d294e24278b2ac751b, parallelism: 2, maxParallelism: 128,
coordinatorState: (none), sub task states: 0, total size (bytes): 0),
OperatorState(operatorID: 9b57d0745f8996333b2b4a42a1758152, parallelism: 4,
maxParallelism: 128, coordinatorState: (none), sub task states: 0, total size
(bytes): 0), OperatorState(operatorID: feca28aff5a3958840bee985ee7de4d3,
parallelism: 1, maxParallelism: 128, coordinatorState: 240 bytes, sub task states:
1, total size (bytes): 921), OperatorState(operatorID:
6e2990595589b48c1ece343fd2076f08, parallelism: 4, maxParallelism: 128,
coordinatorState: (none), sub task states: 4, total size (bytes): 4580),
OperatorState(operatorID: 605b35e407e90cda15ad084365733fdd, parallelism: 1,
maxParallelism: 128, coordinatorState: 232 bytes, sub task states: 1, total size
(bytes): 905), OperatorState(operatorID: e52582678d5fdb8e13f3e6adf9a0d271,
```

```
parallelism: 4, maxParallelism: 128, coordinatorState: (none), sub task states: 4,
total size (bytes): 188528), OperatorState(operatorID:
001a3bdd6238da7f5463f60c314d46ef, parallelism: 4, maxParallelism: 128,
coordinatorState: (none), sub task states: 0, total size (bytes): 0),
OperatorState(operatorID: 313b9f8f24cdfb45b6c03407e78e19ab, parallelism: 4,
maxParallelism: 128, coordinatorState: (none), sub task states: 4, total size
(bytes): 16724), OperatorState(operatorID: 9aeaba2e9dcf4ef63580bd4c145419db,
parallelism: 4, maxParallelism: 128, coordinatorState: (none), sub task states: 4,
total size (bytes): 5176)
2022-09-29 07:08:33,312 INFO  [,] [SourceCoordinator-Source: operator source:read
white list from kafka]
org.apache.flink.runtime.source.coordinator.SourceCoordinator:[265] - Marking
checkpoint 1 as completed for source Source: operator source:read white list from
kafka.
2022-09-29 07:08:33,312 INFO  [,] [SourceCoordinator-Source: operator source:read
config white list from kafka]
org.apache.flink.runtime.source.coordinator.SourceCoordinator:[265] - Marking
checkpoint 1 as completed for source Source: operator source:read config white
list from kafka.
2022-09-29 07:08:33,313 INFO  [,] [SourceCoordinator-Source: operator source:read
threat log from kafka -> Sink: sink : sink to etl]
org.apache.flink.runtime.source.coordinator.SourceCoordinator:[265] - Marking
checkpoint 1 as completed for source Source: operator source:read threat log from
kafka -> Sink: sink : sink to etl.
```