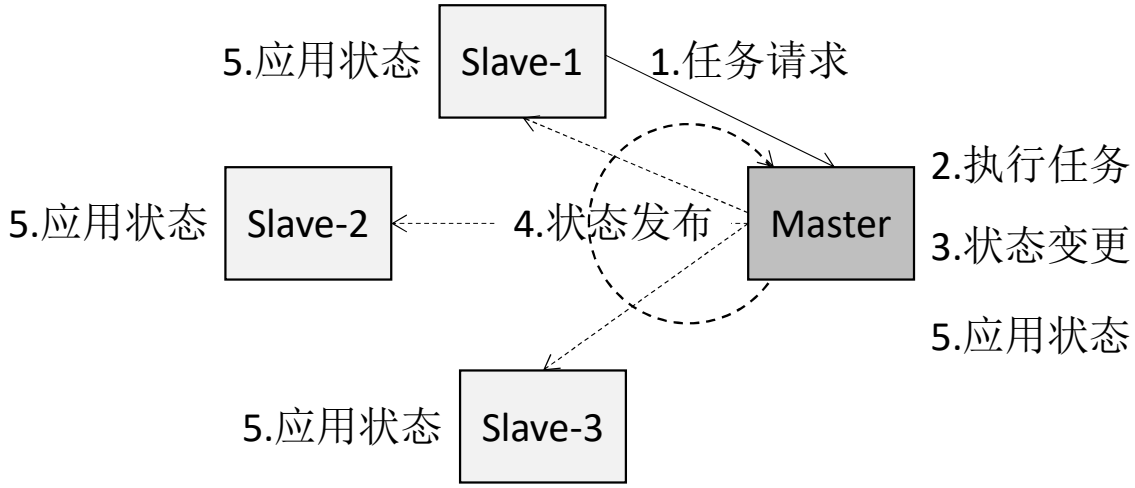
ES集群任务与状态管理

Cluster模块封装了执行集群状态变更任务以及发布、应用集群状态的服务和常用操作。

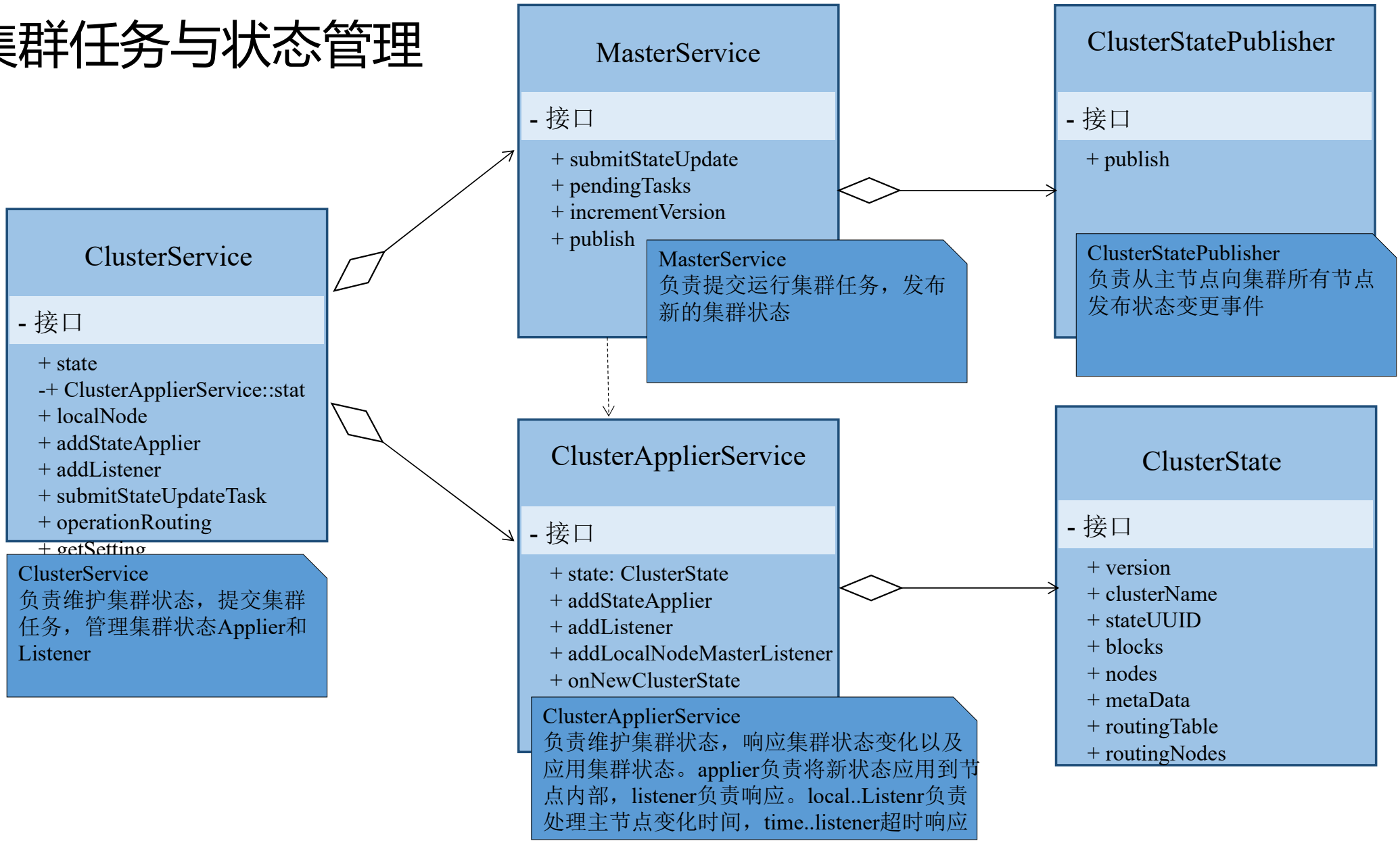
- 集群节点管理
- 分片迁移管理
- 集群状态、元信息管理
- 其他涉及状态变更的任务管理



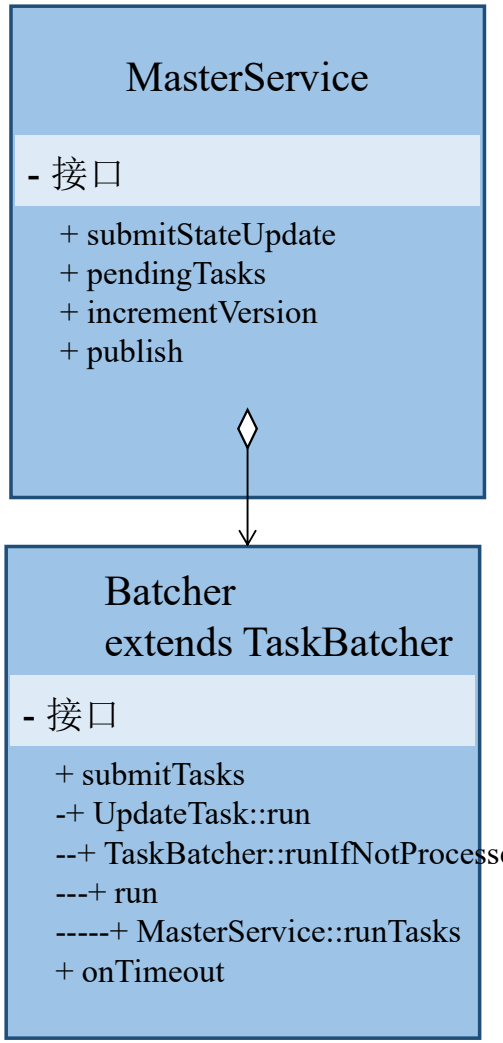
涉及集群层面变更的任务需要由主节点执行，并将新产生的集群状态广播到其他节点。

1、集群拓扑变化
2、模板、索引map、别名变化
3、索引操作
4、pipeline增删
5、脚本增删
6、分片分配
...

ES集群任务与状态管理



集群任务运行管理：提交任务

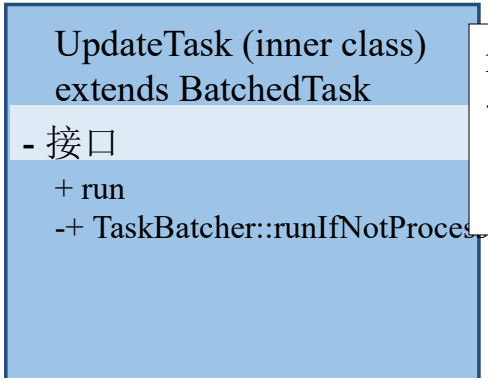


submitStateUpdate(config, executor, listener)
--- 提交集群任务
--- config: 超时时间、优先级
--- executor --> ClusterStateTaskExecutor: 任务具体操作
--- listener: 响应失败、状态处理事件

Batcher::submitTasks 提交任务

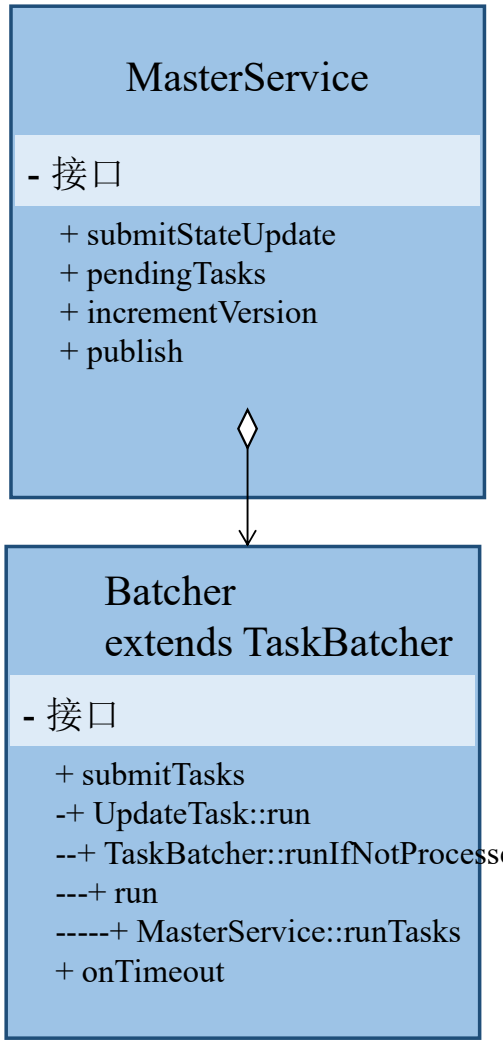
```
public <T extends ClusterStateTaskConfig & ClusterStateTaskExecutor<T> & ClusterStateTaskListener>
    void submitStateUpdateTask(
    String source, T updateTask) {
    submitStateUpdateTask(source, updateTask, updateTask, updateTask, updateTask);
}
```

```
List<Batcher.UpdateTask> safeTasks = tasks.entrySet().stream()
    .map(e -> taskBatcher.new UpdateTask(config.priority(), source, e.getKey(), safe(e.getValue(), supplier), executor))
    .collect(Collectors.toList());
taskBatcher.submitTasks(safeTasks, config.timeout());
```



runIfNotProcessed(BatchedTask updateTask)

集群任务运行管理：提交任务



submitStateUpdate(config, executor, listener)
--- 提交集群任务
--- config: 超时时间、优先级
--- executor --> ClusterStateTaskExecutor: 任务具体操作
--- listener: 响应失败、状态处理事件

```
public <T extends ClusterStateTaskConfig & ClusterStateTaskExecutor<T> & ClusterStateTaskListene
    void submitStateUpdateTask(
    String source, T updateTask) {
    submitStateUpdateTask(source, updateTask, updateTask, updateTask, updateTask);
}
```

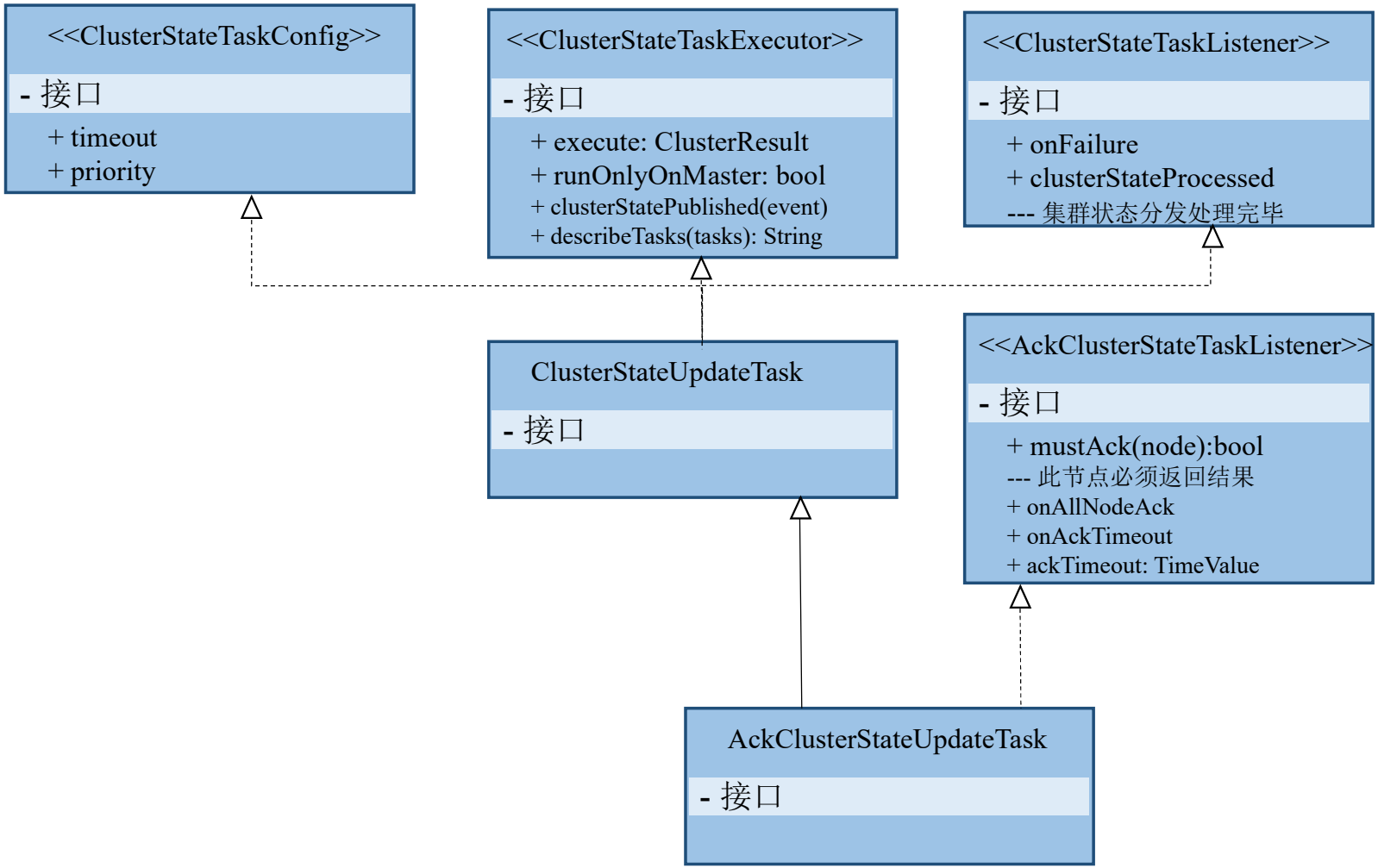
Batcher::submitTasks 提交任务
--- 任务去重，具有相同executor的任务均为重复任务
--- 相同任务以executor为key，存于List中
--- 开始执行第一个任务

```
List<Batcher.UpdateTask> safeTasks = tasks.entrySet().stream()
    .map(e -> taskBatcher new UpdateTask(config.priority(), source, e.getKey(), safe(e.getValue(), supplier), executor))
    .collect(Collectors.toList());
taskBatcher.submitTasks(safeTasks, config.timeout());
```

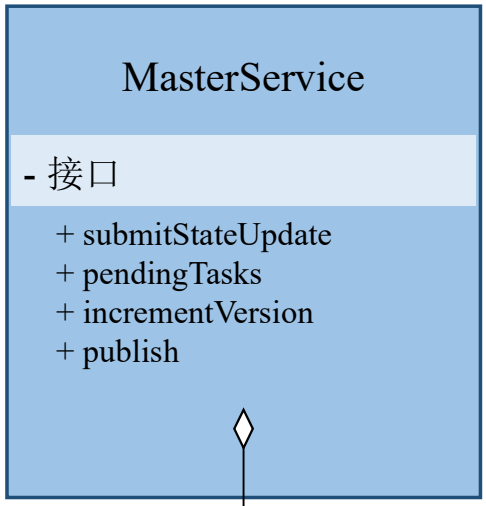
UpdateTask (inner class)
extends BatchedTask
- 接口
+ run
-+ TaskBatcher::runIfNotProce

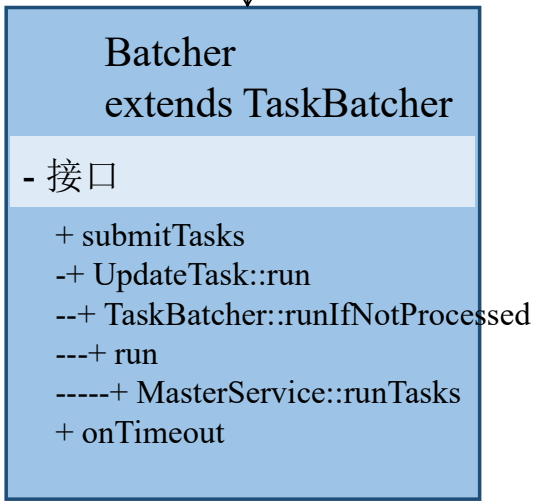
runIfNotProcessed(BatchedTask updateTask)
--- 获取第一个任务的executor
--- 以executor为key，获取其余具有相同executor的任务
--- 执行 Batcher::run
--- 调用MasterService::runTasks，获取任务结果

```
<T extends ClusterStateTaskConfig & ClusterStateTaskExecutor<T> & ClusterStateTaskListener>
T updateTask)
```



集群任务运行管理：





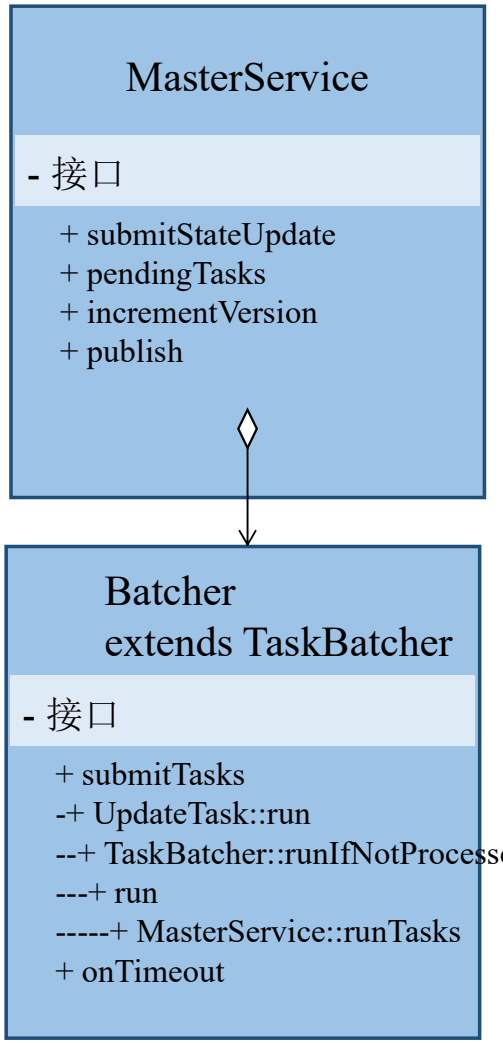

```
public void submitTasks(List<? extends BatchedTask> tasks, @Nullable TimeValue timeout) throws EsRejectedExecutionException
    if (tasks.isEmpty()) {...}
    final BatchedTask firstTask = tasks.get(0);
    assert tasks.stream().allMatch(t -> t.batchingKey == firstTask.batchingKey) :
        "tasks submitted in a batch should share the same batching key: " + tasks;
    // convert to an identity map to check for dups based on task identity
    final Map<Object, BatchedTask> tasksIdentity = tasks.stream().collect(Collectors.toMap(
        BatchedTask::getTask,
        Function.identity(),
        (a, b) -> { throw new IllegalStateException("cannot add duplicate task: " + a); },
        IdentityHashMap::new));

    synchronized (tasksPerBatchingKey) {
        LinkedHashSet<BatchedTask> existingTasks = tasksPerBatchingKey.computeIfAbsent(firstTask.batchingKey,
            k -> new LinkedHashSet<>(tasks.size()));
        for (BatchedTask existing : existingTasks) {
            // check that there won't be two tasks with the same identity for the same batching key
            BatchedTask duplicateTask = tasksIdentity.get(existing.getTask());
            if (duplicateTask != null) {
                throw new IllegalStateException("task [" + duplicateTask.describeTasks(
                    Collections.singletonList(existing)) + "] with source [" + duplicateTask.source + "] is already queued");
            }
        }
        existingTasks.addAll(tasks);
    }

    if (timeout != null) {
        threadExecutor.execute(firstTask, timeout, () -> onTimeoutInternal(tasks, timeout));
    } else {
        threadExecutor.execute(firstTask);
    }
}
```

提交任务去重

任务队列去重

每个任务均绑定有Listener，

集群任务运行管

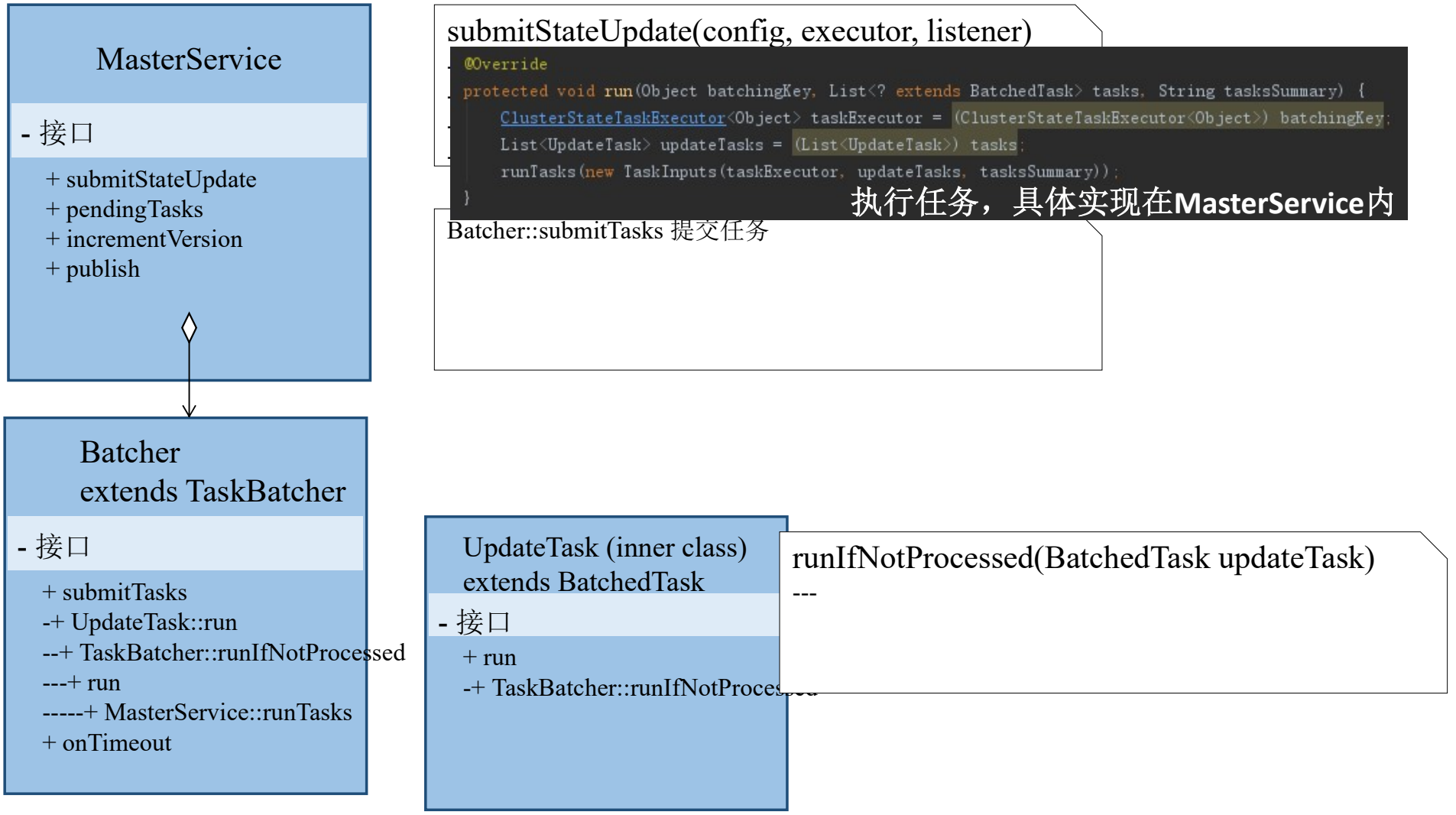



```
void runIfNotProcessed(BatchedTask updateTask) {
    // if this task is already processed, it shouldn't execute other tasks with same batching key that arrived
    // to give other tasks with different batching key a chance to execute.
    if (updateTask.processed.get() == false) {
        final List<BatchedTask> toExecute = new ArrayList<>();
        final Map<String, List<BatchedTask>> processTasksBySource = new HashMap<>();
        synchronized (tasksPerBatchingKey) {
            LinkedHashSet<BatchedTask> pending = tasksPerBatchingKey.remove(updateTask.batchingKey);
            if (pending != null) {
                for (BatchedTask task : pending) {
                    if (task.processed.getAndSet( newValue: true) == false) {
                        logger.trace( message: "will process {}", task);
                        toExecute.add(task);
                        processTasksBySource.computeIfAbsent(task.source, s -> new ArrayList<>()).add(task);
                    } else {
                        logger.trace( message: "skipping {}, already processed", task);
                    }
                }
            }
        }

        if (toExecute.isEmpty() == false) {
            final String tasksSummary = processTasksBySource.entrySet().stream().map(entry -> {
                String tasks = updateTask.describeTasks(entry.getValue());
                return tasks.isEmpty() ? entry.getKey() : entry.getKey() + "[" + tasks + "]";
            }).reduce((s1, s2) -> s1 + ", " + s2).orElse( other: "");

            run(updateTask.batchingKey, toExecute, tasksSummary);
        }
    }
```

获取有相同executor的所有

执行任务

集群任务运行管理：提交任务



集群任务运行管

MasterService

- 接口

+ submitStateUpdate
+ pendingTasks
+ incrementVersion
+ publish

Batcher
extends TaskBatcher

- 接口

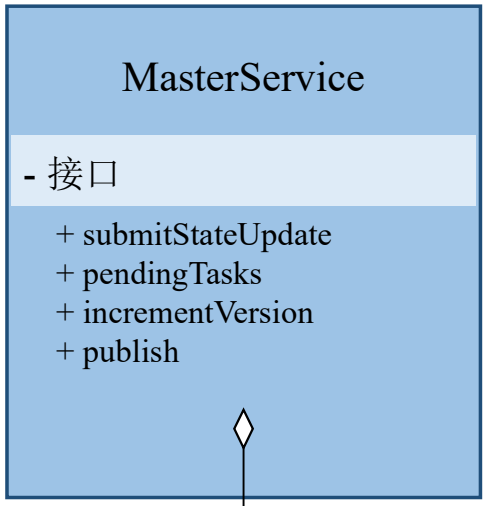
+ submitTasks
-+ UpdateTask::run
--+ TaskBatcher::runIfNotProces
---+ run
-----+ MasterService::runTasks
+ onTimeout

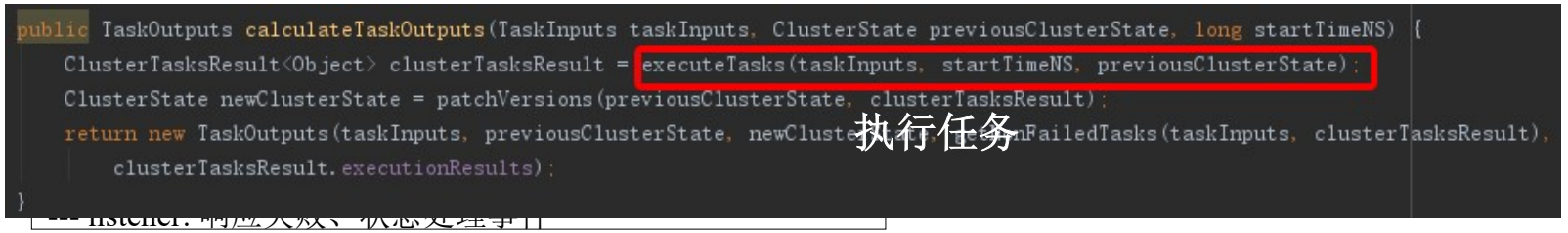
```
protected void runTasks(TaskInputs taskInputs) {
    final String summary = taskInputs.summary;
    if (!lifecycle.started()) {...}

    logger.debug( message: "processing [{}]: execute", summary);
    final ClusterState previousClusterState = state();

    if (!previousClusterState.nodes().isLocalNodeElectedMaster() && taskInputs.runOnlyWhenMaster()) {...}

    long startTimeNS = currentTimeInNanos();
    TaskOutputs taskOutputs = calculateTaskOutputs(taskInputs, previousClusterState, startTimeNS);
    taskOutputs.notifyFailedTasks();

    if (taskOutputs.clusterStateUnchanged()) {
        taskOutputs.notifySuccessfulTasksOnUnchangedClusterState();
        TimeValue executionTime = TimeValue.timeValueMillis(Math.max(0, TimeValue.nsecToMSec(currentTimeInNanos() - startTimeNS)));
        logger.debug( message: "processing [{}]: took [{}] no change in cluster state", summary, executionTime);
        warnAboutSlowTaskIfNeeded(executionTime, summary);
    } else {
        ClusterState newClusterState = taskOutputs.newClusterState;
        if (logger.isTraceEnabled()) {...} else if (logger.isDebugEnabled()) {...}
        try {
            ClusterChangedEvent clusterChangedEvent = new ClusterChangedEvent(summary, newClusterState, previousClusterState);
            // new cluster state, notify all listeners
            final DiscoveryNodes.Delta nodesDelta = clusterChangedEvent.nodesDelta();
            if (nodesDelta.hasChanges() && logger.isInfoEnabled()) {...}

            logger.debug( message: "publishing cluster state version [{}]", newClusterState.version());
            publish(clusterChangedEvent, taskOutputs, startTimeNS);
        } catch (Exception e) {...}
    }
}
```

执行任务，获取结果

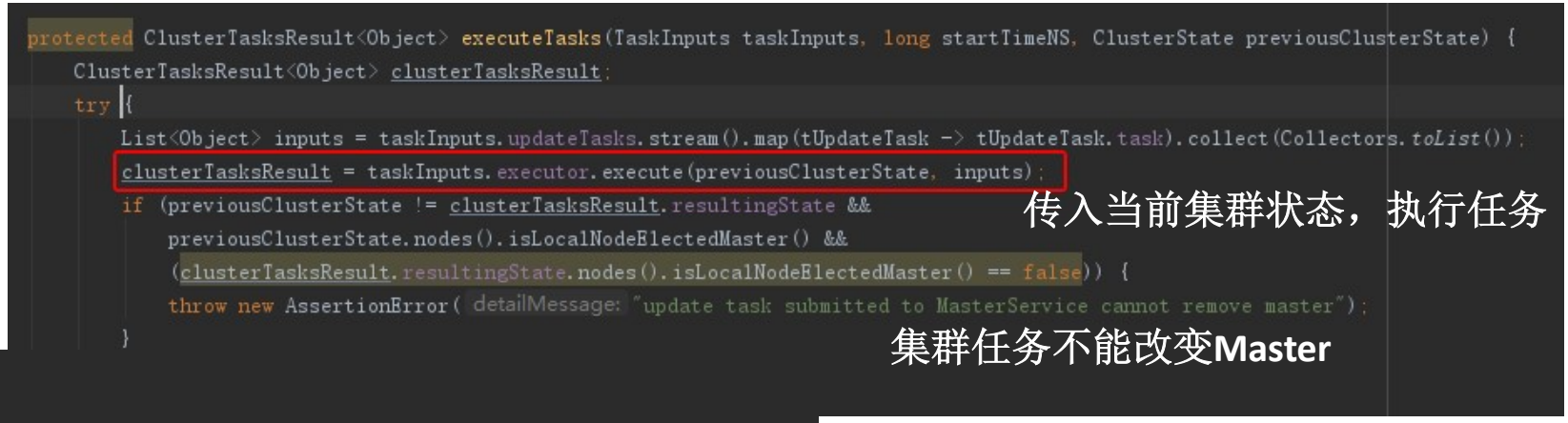
发布集群的新状态

集群任务运行管理：提交任务



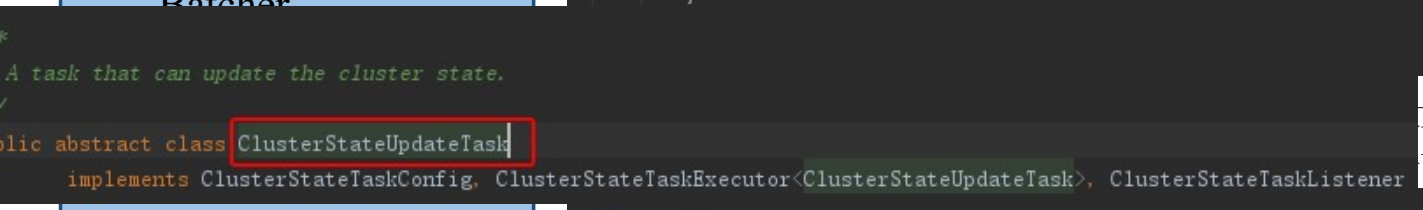


执行任务



传入当前集群状态，执行任务

集群任务不能改变Master



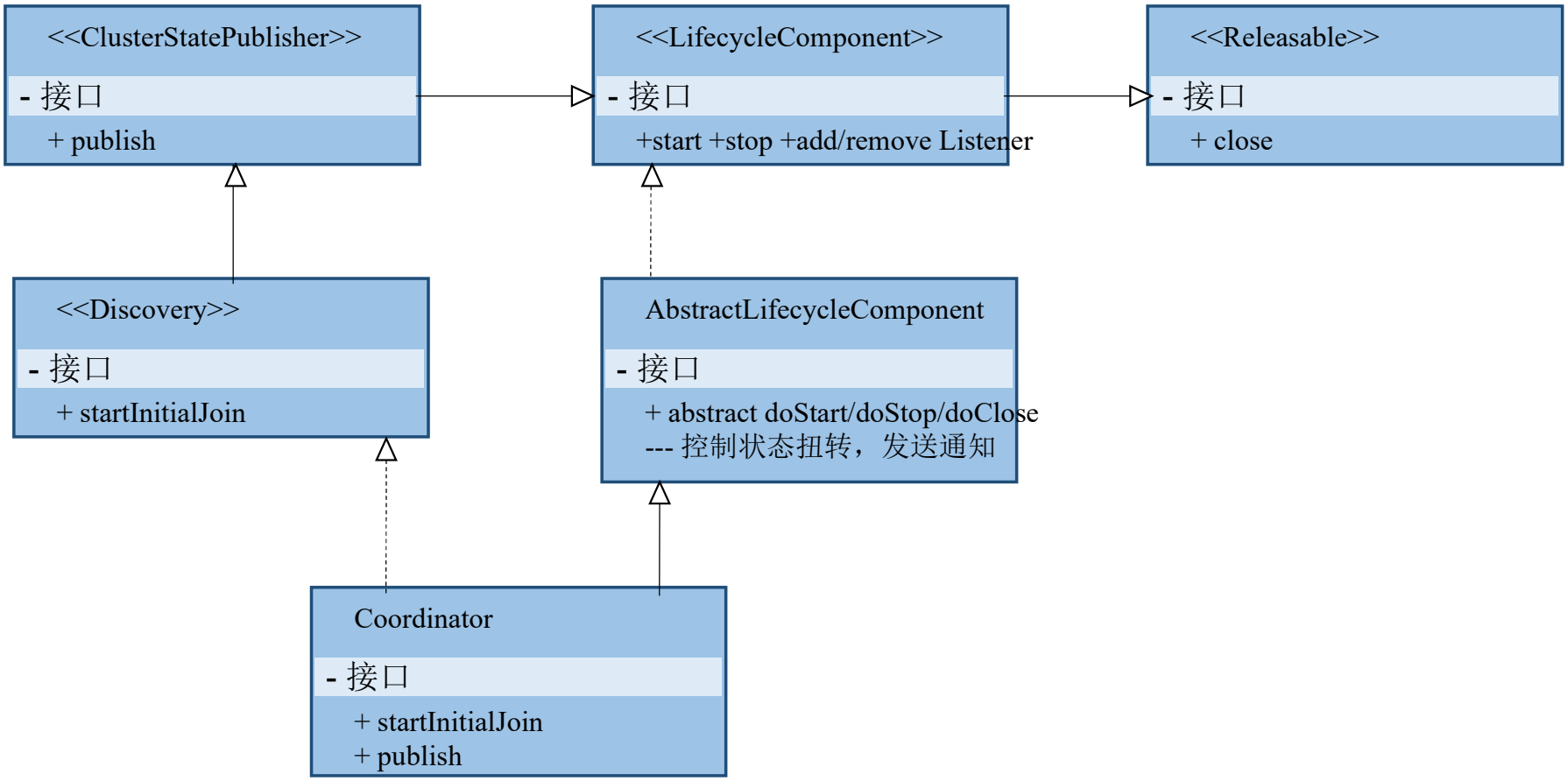
BatchedTask updateTask)

-+ UpdateTask::run
--+ TaskBatcher::runIfNotProcesse
---+ run
-----+ MasterService::runTasks
+ onTimeout

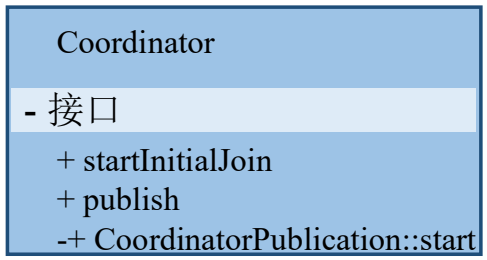
调用任务的具体实现

批量设置重复task的执行结果

集群任务运行管理：集群状态发布

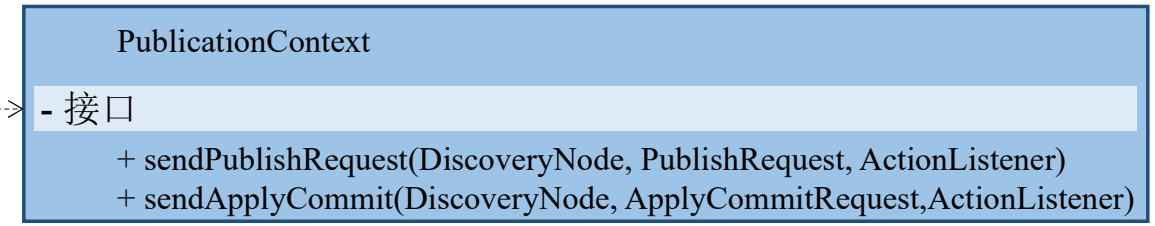


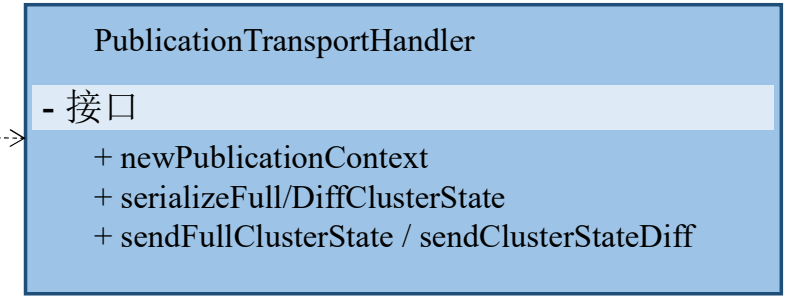
集群状态发布

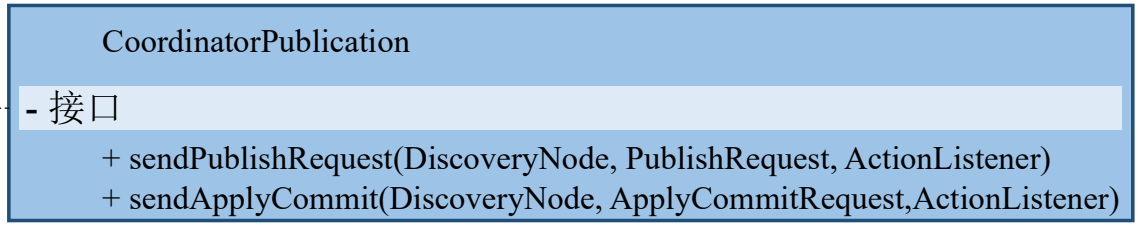


publish(clusterChangeEvent, ActionListener, AckListener)
--- 发布集群状态
--- event: 包含新旧集群状态，集群节点引用
--- ActionListener: 响应返回结果和发布失败事件
--- AckListener: 响应状态提交以及节点成功应用状态事件

构造发布状态所需的环境
PublicationContext(clusterChangedEvent): 构造需发布的全量或增量数据，实现二阶段提交
PublishRequest(clusterState): 持有需要发布的最新集群状态
DiscoveryNodes: ClusterState::nodes: 需发布的节点
CoordinatorPublication(publishRequest, publicationContext, ListenableFuture, ackListener, publishListener)
CoordinatorPublication::start

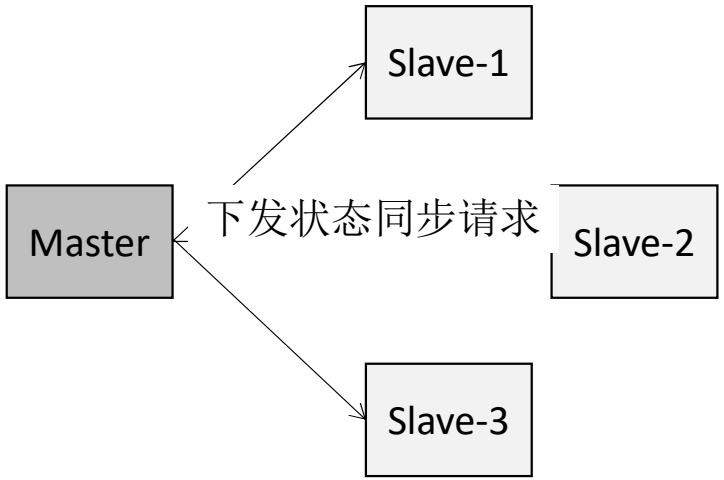






集群状态发布：二阶段提交

避免状态同步失败回滚

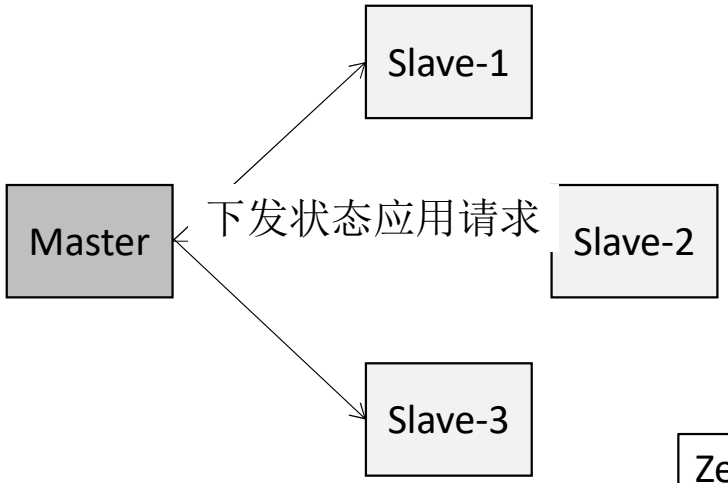


1、第一阶段：发布集群状态，等待响应

discovery.zen.minimum_master_node=3

若接收到minimum_master_node-1=2个节点响应，即认定第一阶段成功

超时未满足成功条件，同步失败
discovery.zen.commit_timeout



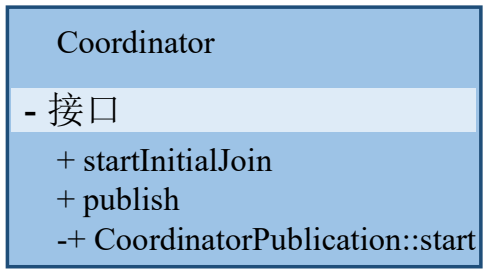
2、第二阶段：发送commit请求，应用状态
若接收到全部节点响应，即认定第二阶段成功

超时未满足成功条件，同步失败
discovery.zen.commit.\
publish_timeout - commit_timeout

状态同步成功

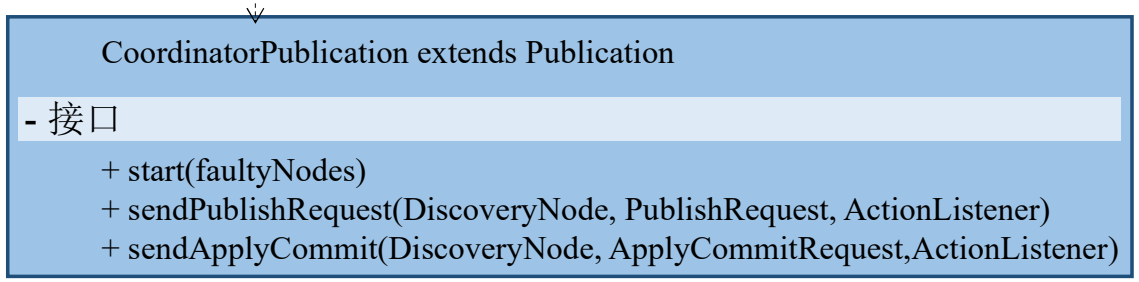
ZenDiscovery::processNextCommittedClusterState

集群状态发布流程：状态同步请求



构造发布状态所需的环境

```
final PublicationTransportHandler.PublicationContext publicationContext =
    publicationHandler.newPublicationContext(clusterChangedEvent);

final PublishRequest publishRequest = coordinationState.get().handleClientValue(clusterState);
final CoordinatorPublication publication = new CoordinatorPublication(publishRequest, publicationContext,
    new ListenableFuture<>(), ackListener, publishListener);
```



从节点连接性和健康检查

```
leaderChecker.setCurrentNodes(publishNodes);
followersChecker.setCurrentNodes(publishNodes);
lagDetector.setTrackedNodes(publishNodes);
```

向每个节点发送状态同步请求 publishRequest

```
public abstract class Publication
    publicationTargets.forEach(PublicationTarget::sendPublishRequest);
```

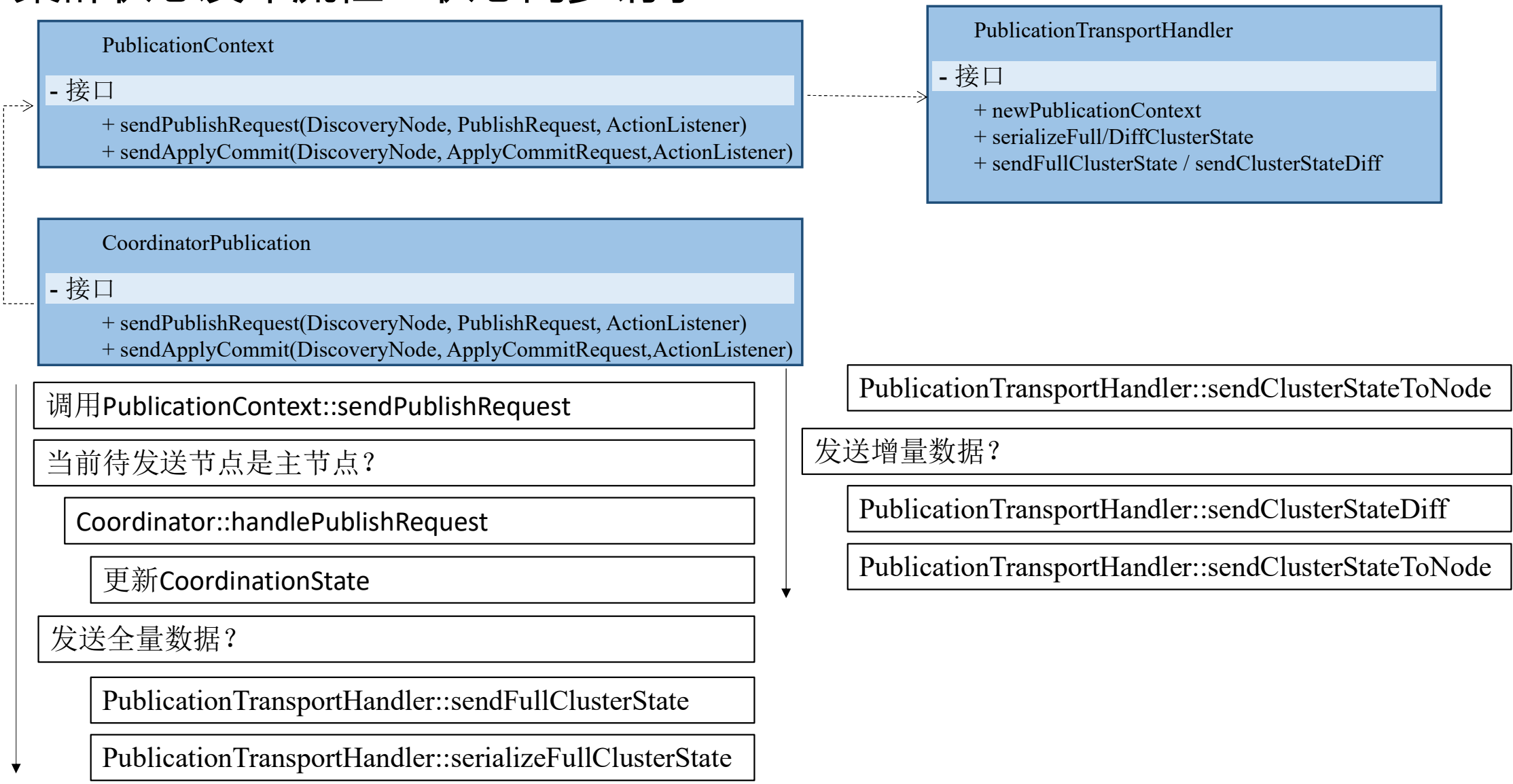
```
class PublicationTarget   void sendPublishRequest()   Publication.this.sendPublishRequest(discoveryNode, publishRequest, new PublishResponseHandler());
```

```
public abstract class Publication   protected abstract void sendPublishRequest
```

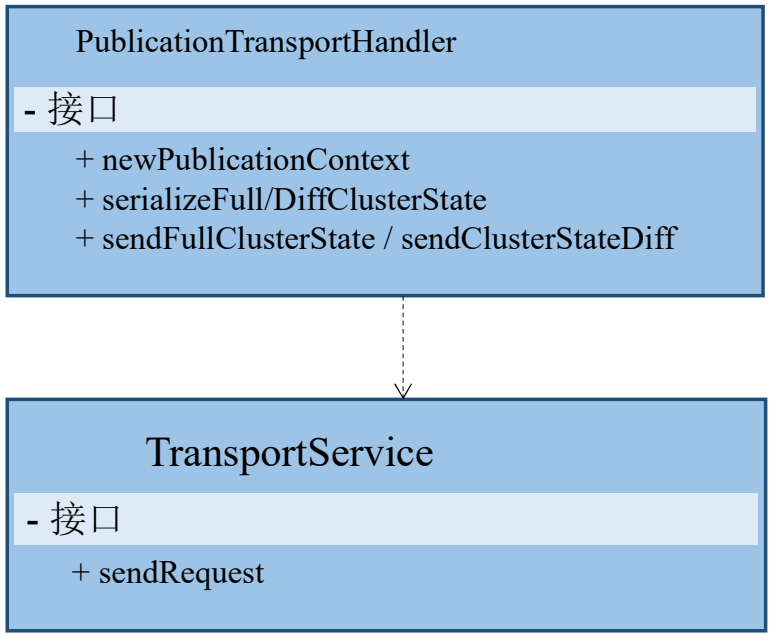
```
class CoordinatorPublication extends Publication
@Override
protected void sendPublishRequest(DiscoveryNode destination, PublishRequest publishRequest,
                                  ActionListener<PublishWithJoinResponse> responseActionListener) {
    publicationContext.sendPublishRequest(destination, publishRequest, wrapWithMutex(responseActionListener));
}
```

调用PublicationContext::sendPublishRequest

集群状态发布流程：状态同步请求



集群状态发布流程：状态同步请求



PublicationTransportHandler::sendClusterStateToNode

Zen1Node? sendRequest to PublishClusterStateAction::SEND_ACTION_NAME

Zen2Node? sendRequest to PublicationTransportHandler::PUBLISH_STATE_ACTION_NAME

从节点侧,接收同步请求并处理

Zen1Node: SEND_ACTION_NAME handler

PublishClusterStateAction::handleIncomingClusterStateRequest

```
public class PublishClusterStateAction
protected void handleIncomingClusterStateRequest
    incomingState = ClusterState.readFrom(in, transportService.getLocalNode());
    fullClusterStateReceivedCount.incrementAndGet();
    Diff<ClusterState> diff = ClusterState.readDiffFrom(in, lastSeenClusterState.nodes().getLocalNode());
    incomingState = diff.apply(lastSeenClusterState);
    compatibleClusterStateDiffReceivedCount.incrementAndGet();
    lastSeenClusterState = incomingState;
    channel.sendResponse(TransportResponse.Empty.INSTANCE);
```

获取最新clusterState(全量或增量), 应用增量来获取全量

保存最新全量clusterState

返回成功Response

集群状态发布流程：状态同步请求

PublicationTransportHandler

- 接口

+ newPublicationContext
+ serializeFull/DiffClusterState
+ sendFullClusterState / sendClusterStateDiff

TransportService

- 接口

+ sendRequest

PublicationTransportHandler::sendClusterStateToNode

Zen1Node? sendRequest to PublishClusterStateAction::SEND_ACTION_NAME

Zen2Node? sendRequest to PublicationTransportHandler::PUBLISH_STATE_ACTION_NAME

从节点侧,接收同步请求并处理

Zen2Node: PUBLISH_STATE_ACTION_NAME handler

PublicationTransportHandler::handleIncomingPublishRequest

```
public class PublicationTransportHandler
private PublishWithJoinResponse handleIncomingPublishRequest(
    incomingState = ClusterState.readFrom(in, transportService.getLocalNode());
    fullClusterStateReceivedCount.incrementAndGet();
    Diff<ClusterState> diff = ClusterState.readDiffFrom(in, lastSeenClusterState.nodes().getLocalNode());
    incomingState = diff.apply(lastSeenClusterState);
    compatibleClusterStateDiffReceivedCount.incrementAndGet();
    lastSeenClusterState = incomingState;
    final PublishWithJoinResponse response = handlePublishRequest.apply(new PublishRequest(incomingState));
```

获取最新clusterState(全量或增量), 应用增量来获取全量

保存最新全量clusterState

返回成功Response

集群状态发布流程：状态应用请求

CoordinatorPublication extends Publication

- 接口
  - + start(faultyNodes)
  - + sendPublishRequest(DiscoveryNode, PublishRequest, ActionListener)
  - + sendApplyCommit(DiscoveryNode, ApplyCommitRequest,ActionListener)

调用Publication::sendPublishRequest

PublishResponseHandler::onResponse

更新响应节点个数，更新状态WAITING_FOR_QUORUM

```
publishVotes.addVote(sourceNode);
if (isPublishQuorum(publishVotes)) {
    logger.trace( message: "handlePublishResponse: value committed for version [{}]
        publishResponse.getVersion(), publishResponse.getTerm());
    return Optional.of(new ApplyCommitRequest(localNode, publishResponse.getTerm(),
}

return Optional.empty();
```

调用Publication::handlePublishResponse

调用CoordinationState::handlePublishResponse

判断是否满足响应个数限制 —— 是

```
publicationTargets.stream().filter(PublicationTarget::isWaitingForQuorum).forEach(PublicationTarget::sendApplyCommit);
```

调用Publication::sendApplyCommit，向所有节点发送

```
if (Coordinator.isZen1Node(destination)) {
    actionName = PublishClusterStateAction.COMMIT_ACTION_NAME;
    transportRequest = new PublishClusterStateAction.CommitClusterStateRequest(newState.stateUUID());
} else {
    actionName = COMMIT_STATE_ACTION_NAME;
    transportRequest = applyCommitRequest;
}
transportService.sendRequest(destination, actionName, transportRequest, stateRequestOptions,
    new TransportResponseHandler<TransportResponse.Empty>() {...});
```

调用PublicationContext::sendApplyCommit

Zen1Node? sendRequest to PublishClusterStateAction::COMMIT_ACTION_NAME

Zen2Node? sendRequest to PublicationTransportHandler::COMMIT_STATE_ACTION_NAME

从节点侧,接收状态应用请求并处理

集群状态发布流程：状态应用请求

从节点侧,接收状态应用请求并处理

Zen1Node: COMMIT_ACTION_NAME handler

PublishClusterStateAction::handleCommitRequest

ZenDiscovery::onClusterStateCommitted

ZenDiscovery::processNextCommittedClusterState

更新committedState，记录最近提交的集群状态

集群主/从节点信息校正

ClusterApplierService::onNewClusterState

通知Appliers应用集群状态，更新集群状态

通知Listeners响应集群状态改变事件

返回成功Response

```
public class ZenDiscovery public void onClusterStateCommitted(
processNextCommittedClusterState( reason: "master " + state.nodes().getMasterNode() +
    " committed version [" + state.version() + "]");

// return true if state has been sent to applier
boolean processNextCommittedClusterState(String reason)
                committedState.set(newClusterState);
if (newClusterState.nodes().isLocalNodeElectedMaster()) nodesFD.updateNodesAndPing(newClusterState);
else {
masterFD.restart(newClusterState.nodes().getMasterNode(),
    reason: "new cluster state received and we are monitoring the wrong master [" + masterFD.masterNode() + "]");

public class ClusterApplierService public void onNewClusterState(
  submitStateUpdateTask(source, ClusterStateTaskConfig.build(Priority.HIGH), applyFunction, listener);
    applyChanges(task, previousClusterState, newClusterState);
    private void applyChanges(
      callClusterStateAppliers(clusterChangedEvent);
      state.set(newClusterState);
      callClusterStateListeners(clusterChangedEvent);
```

集群状态发布流程：状态应用请求

从节点侧,接收状态应用请求并处理

Zen2Node: COMMIT_STATE_ACTION_NAME handler

Coordinator::handleApplyCommit

ClusterApplierService::onNewClusterState

通知Appliers应用集群状态，更新集群状态

通知Listeners响应集群状态改变事件

返回成功Response

```
public class PublicationTransportHandler   public PublicationTransportHandler
```

```
transportService.registerRequestHandler(COMMIT_STATE_ACTION_NAME, ThreadPool.Names.GENERIC,
    ApplyCommitRequest::new,
    (request, channel, task) -> handleApplyCommit.accept(request, transportCommitCallback(ch
```

```
public class Coordinator   private void handleApplyCommit
```

```
if (applyCommitRequest.getSourceNode().equals(getLocalNode())) {
    // master node applies the committed state at the end of the publication process, no
    applyListener.onResponse(null);
} else {
    clusterApplier.onNewClusterState(applyCommitRequest.toString(), () -> applierState,
        new ClusterApplyListener() {...});
}
```

```
public class PublicationTransportHandler   private ActionListener<Void> transportCommitCallback(TransportChannel channel)
```

```
return new ActionListener<Void>() {

    @Override
    public void onResponse(Void aVoid) {
        try {
            channel.sendResponse(TransportResponse.Empty.INSTANCE);
```

集群状态发布：全量/增量发布

强制全量同步或者节点上次同步不成功

↓ 准备全量数据

序列化集群状态并压缩

```
public static BytesReference serializeFullClusterState(ClusterState clusterState, Vers
    final BytesStreamOutput bStream = new BytesStreamOutput();
    try (StreamOutput stream = CompressorFactory.COMPRESSOR.streamOutput(bStream)) {
        stream.setVersion(nodeVersion);
        stream.writeBoolean( b: true);
        clusterState.writeTo(stream);
    }
    return bStream.bytes();
}
```

不强制全量同步且者节点上次同步成功

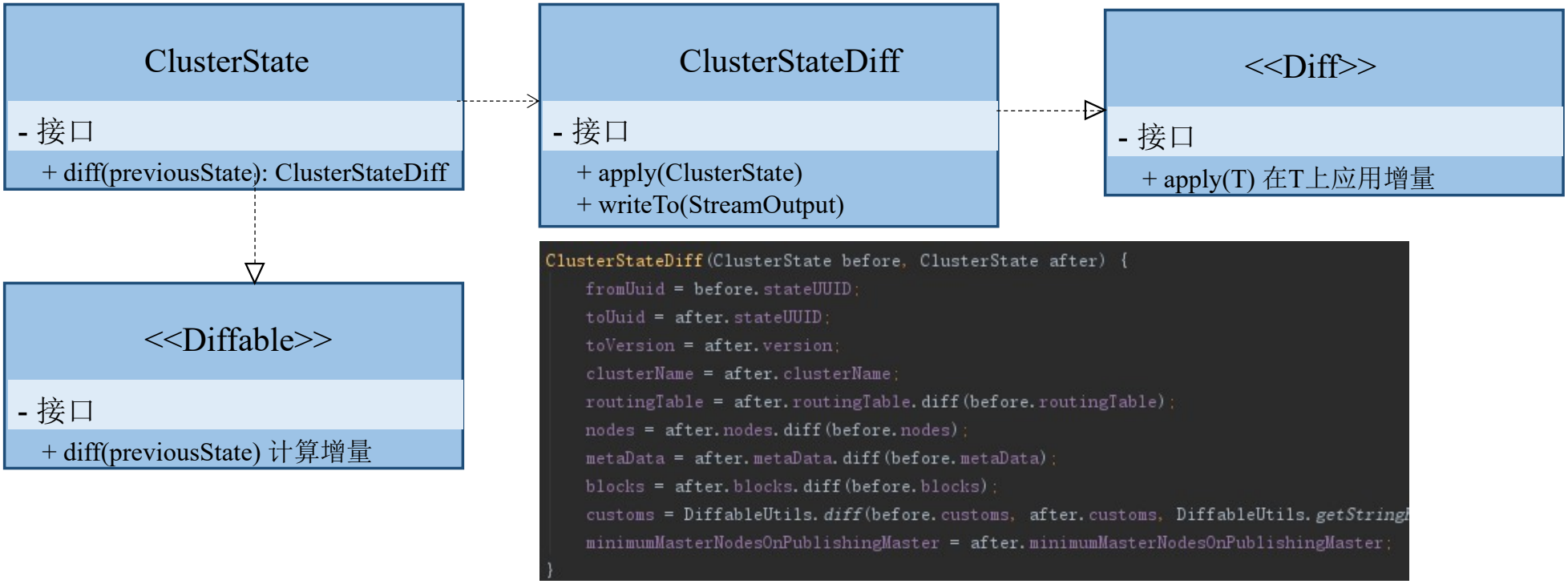
↓ 准备增量数据

计算增量，序列化并压缩

```
if (diff == null) {
    diff = clusterState.diff(previousState);
}
```

```
public static BytesReference serializeDiffClusterState(Diff diff, Version nodeVersion
    final BytesStreamOutput bStream = new BytesStreamOutput();
    try (StreamOutput stream = CompressorFactory.COMPRESSOR.streamOutput(bStream)) {
        stream.setVersion(nodeVersion);
        stream.writeBoolean( b: false);
        diff.writeTo(stream);
    }
    return bStream.bytes();
}
```

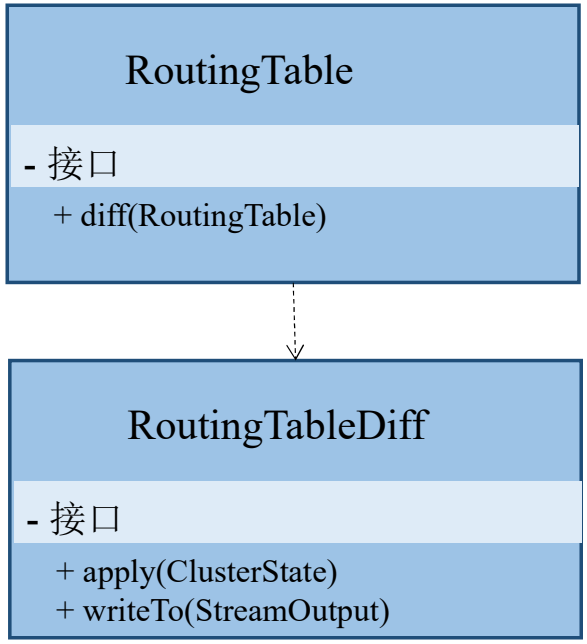
集群状态发布：计算增量




```
ClusterStateDiff(ClusterState before, ClusterState after) {
    fromUuid = before.stateUUID;
    toUuid = after.stateUUID;
    toVersion = after.version;
    clusterName = after.clusterName;
    routingTable = after.routingTable.diff(before.routingTable);
    nodes = after.nodes.diff(before.nodes);
    metaData = after.metaData.diff(before.metaData);
    blocks = after.blocks.diff(before.blocks);
    customs = DiffableUtils.diff(before.customs, after.customs, DiffableUtils.getStringI
    minimumMasterNodesOnPublishingMaster = after.minimumMasterNodesOnPublishingMaster;
}
```

RoutingTable，DiscoveryNodes， MetaData， ClusterBlocks，自定义参数ImmutableOpenMap<String, Custom>>均实现Diffable接口，能够自行计算增量

计算增量：RoutingTable



维护ImmutableOpenMap<String, IndexRoutingTable> indicesRouting

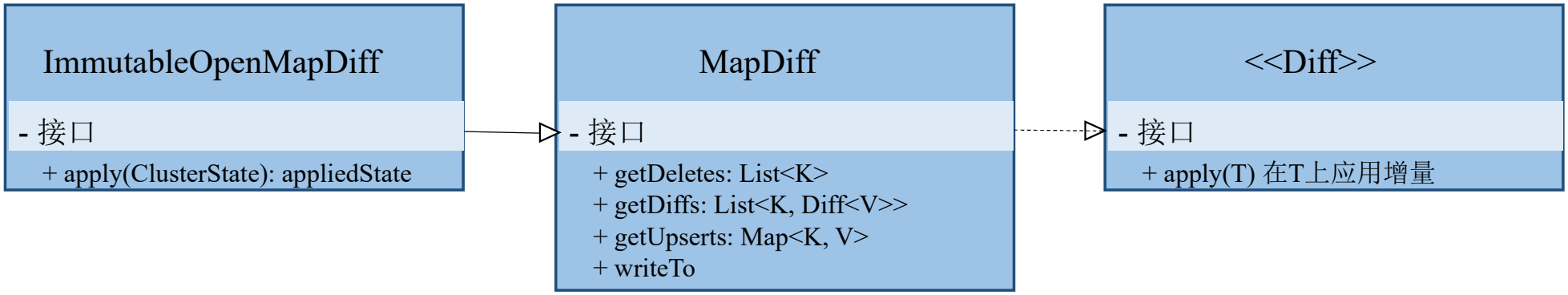
维护Diff<ImmutableOpenMap<String, IndexRoutingTable>> indicesRouting

```
RoutingTableDiff(RoutingTable before, RoutingTable after) {
    version = after.version;
    indicesRouting = DiffableUtils.diff(before.indicesRouting, after.indicesRouting,
}
```

```
public final class DiffableUtils
```

```
public static <K, T extends Diffable<T>> MapDiff<K, T, ImmutableOpenMap<K, T>> diff(ImmutableOpenMap<K, T> before,
        ImmutableOpenMap<K, T> after, KeySerializer<K> keySerializer) {
    assert after != null && before != null;
    return new ImmutableOpenMapDiff<>(before, after, keySerializer, DiffableValueSerializer.getWriteOnlyInstance());
}
```

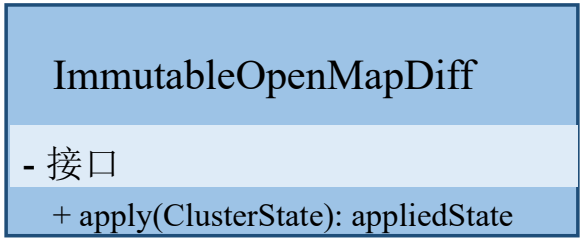
计算增量：ImmutableOpenMapDiff




```
 * @param <K> the type of map keys
 * @param <T> the type of map values
 * @param <M> the map implementation type
 */
public abstract static class MapDiff<K, T, M> implements Diff<M>
```

```
protected final List<K> deletes;
protected final Map<K, Diff<T>> diffs; // incremental updates
protected final Map<K, T> upserts; // additions or full updates
protected final KeySerializer<K> keySerializer;
protected final ValueSerializer<K, T> valueSerializer;
```

计算增量：ImmutableOpenMapDiff




```
public ImmutableOpenMapDiff(ImmutableOpenMap<K, T> before, ImmutableOpenMap<K, T> after,
                            KeySerializer<K> keySerializer, ValueSerializer<K, T> valueSerializer) {
    super(keySerializer, valueSerializer);
    assert after != null && before != null;

    for (ObjectCursor<K> key : before.keys()) {
        if (!after.containsKey(key.value)) {
            deletes.add(key.value);
        }
    }

    for (ObjectObjectCursor<K, T> partIter : after) {
        T beforePart = before.get(partIter.key);
        if (beforePart == null) {
            upserts.put(partIter.key, partIter.value);
        } else if (partIter.value.equals(beforePart) == false) {
            if (valueSerializer.supportsDiffableValues()) {
                diffs.put(partIter.key, valueSerializer.diff(partIter.value, beforePart));
            } else {
                upserts.put(partIter.key, partIter.value);
            }
        }
    }
}
```

计算增量： DiscoveryNodes,MetaData

```
nodes = after.nodes.diff(before.nodes);
```

```
@Override
public Diff<T> diff(T previousState) {
    if (this.get().equals(previousState)) {
        return new CompleteDiff<>();
    } else {
        return new CompleteDiff<>(get());
    }
}
```

```
metaData = after.metaData.diff(before.metaData);
```

```
private Diff<ImmutableOpenMap<String, IndexMetaData>> indices;
private Diff<ImmutableOpenMap<String, IndexTemplateMetaData>> templates;
private Diff<ImmutableOpenMap<String, Custom>> customs;

MetaDataDiff(MetaData before, MetaData after) {
    clusterUUID = after.clusterUUID;
    clusterUUIDCommitted = after.clusterUUIDCommitted;
    version = after.version;
    coordinationMetaData = after.coordinationMetaData;
    transientSettings = after.transientSettings;
    persistentSettings = after.persistentSettings;
    indices = DiffableUtils.diff(before.indices, after.indices, DiffableUtils.getStringKeySerializer());
    templates = DiffableUtils.diff(before.templates, after.templates, DiffableUtils.getStringKeySerializer());
    customs = DiffableUtils.diff(before.customs, after.customs, DiffableUtils.getStringKeySerializer(), CUSTOM_VALUE_SERIALIZER);
}
```