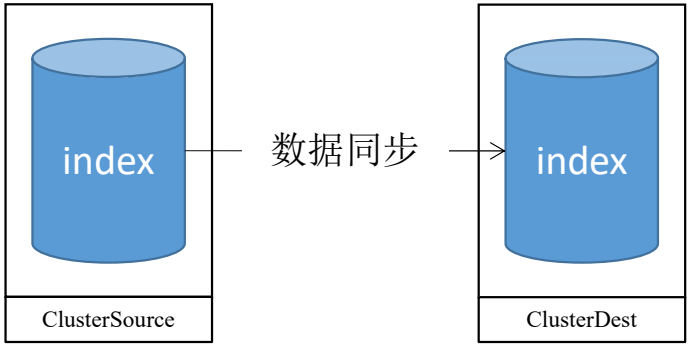
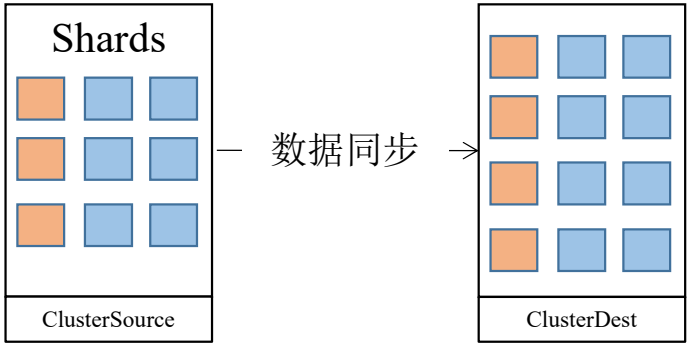
目 录

Reindex简要介绍



reindex负责从源索引查出数据，将数据索引进目标索引。源索引setting、mapping均不会同步



可用于变更索引主分片设置

```
POST _reindex
{
  "source": {
    "remote": {
      "host": "http://otherhost:9200",
      "username": "user",
      "password": "pass"
    },
    "index": "source",
    "query": {
      "match": {
        "test": "data"
      }
    }
  },
  "dest": {
    "index": "dest"
  }
}
```

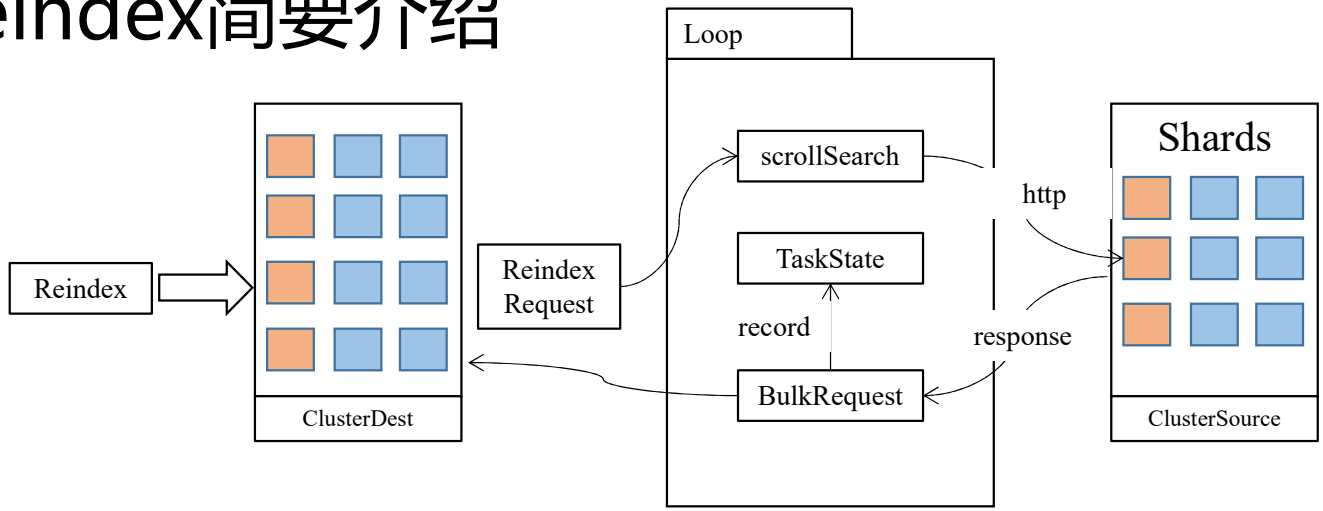
Destination文档版本控制：version_type
1、internal：覆盖所有id相同的文档，忽略版本冲突
2、external或external_gt：只覆盖版本更低的文档
3、external_gte：只覆盖版本相同或更低的文档

```
"dest": {
  "index": "new_twitter",
  "version_type": "external"
},
```

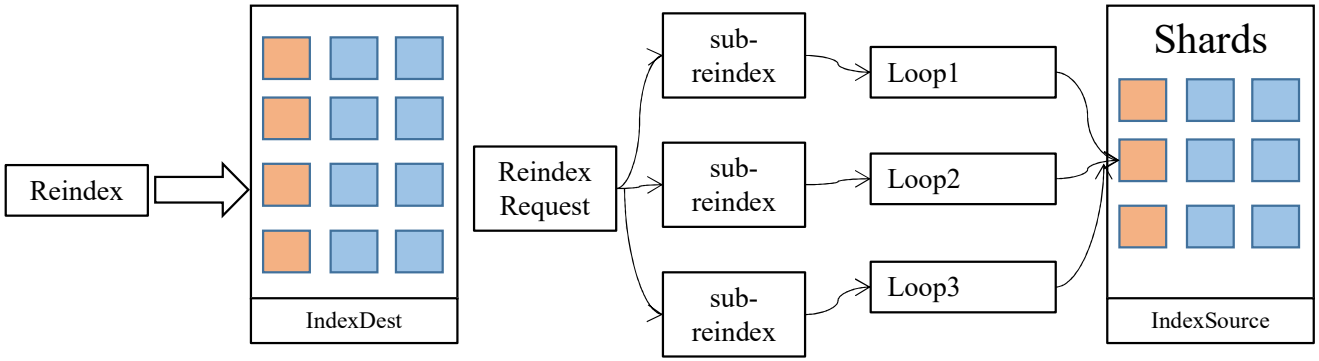
Destination文档冲突产生行为：op_type
1、index：默认行为，不产生冲突
2、create：只创建不存在的文档，所有已存在文档均产生冲突

Reindex文档冲突处理策略：conflicts
1、proceed：忽略文档冲突冲突
2、abort：默认行为，终止reindex任务

Reindex简要介绍



1、跨集群reindex不支持任务切分
2、reindex接收scrollResponse与发送bulkRequest均发生在一个节点上



集群内reindex，可使用scrollSearch的slice功能，将scrollSearch切分为多个

Reindex实现：RestReindexAction

从RequestBody解析构建出ReindexRequest

解析QueryParameters，填充进ReindexRequest

scroll控制context保存时间

解析refresh、timeout

解析slice，任务拆分数

requests_per_second，reindex速率

每批数据预计同步时间 (target_time) = BatchSize / requests_per_seconds
每批数据实际同步时间 = write_time
每批数据间隔等待时间 = target_time - write_time

根据wait_for_completion控制任务同步/异步执行

RestReindexAction::prepareRequest
--- 解析请求

RestReindexAction::buildRequest

解析RequestBody >>> ReindexRequest

```
ObjectParser<ReindexRequest, Void> PARSER = new ObjectParser<>( name: "reindex");
```

```
ReindexRequest internal = new ReindexRequest();
try (XContentParser parser = request.contentParser()) {
    PARSER.parse(parser, internal, context: null);
}
```

scrollSearch Context保留时间

```
if (request.hasParam( key: "scroll")) {
    internal.setScroll(parseTimeValue(request.param( key: "scroll"),
        settingName: "scroll"));
}
```

AbstractBaseReindexRestHandler::setCommonOptions

QueryParameters:refresh, timeout >>> ReindexRequest

```
request.setRefresh(restRequest.paramAsBoolean( key: "refresh", request.isRefresh()));
request.setTimeout(restRequest.paramAsTime( key: "timeout", request.getTimeout()));
```

QueryParameters:slice >>> ReindexRequest

```
String slicesString = request.param( key: "slices");
```

```
slices = Integer.parseInt(slicesString);
```

```
request.setSlices(slices);
```

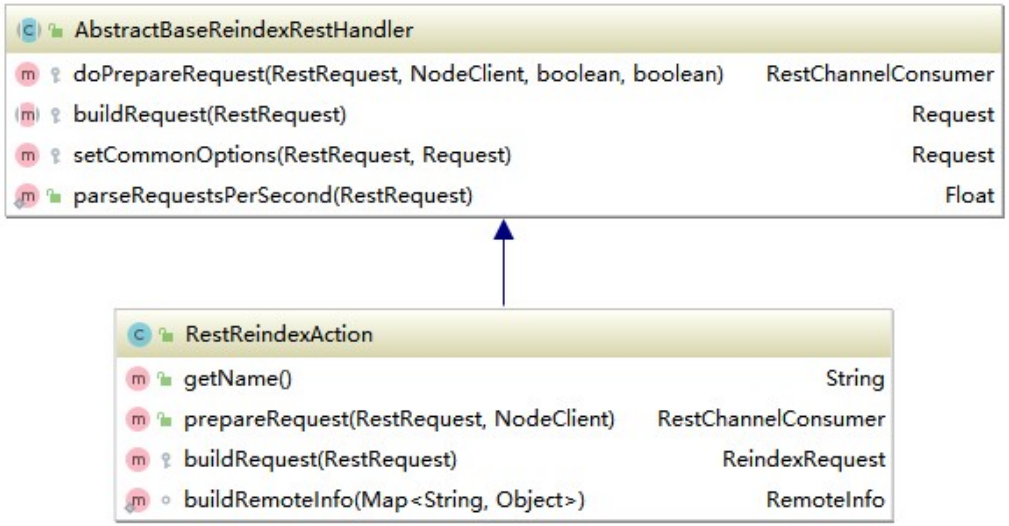
QueryParameters:requests_per_second >>> ReindexRequest

```
String requestsPerSecondString = request.param( key: "requests_per_second");
```

```
requestsPerSecond = Float.parseFloat(requestsPerSecondString);
```

```
request.setRequestsPerSecond(requestsPerSecond);
```

Reindex实现：同步异步控制



RestReindexAction::prepareRequest
AbstractBaseReindexRestHandler::doPrepareRequest
--- 解析请求，执行reindex任务

AbstractBaseReindexRestHandler::doPrepareRequest

根据wait_for_completion确定任务执行方式

```
// Executes the request and waits for completion
if (request.paramAsBoolean( key: "wait_for_completion", defaultValue: true)) {
```

同步方式

NodeClient调用TransportReindexAction执行任务

scrollSearch　　bulkInsert

Node1　Node2　Node1　Node2

finished

调用者收到结果

Listener::onResponse 编码reindex结果
RestChannel::sendResponse 发送结果

异步方式

收到任务Id

发送task标识
localNodeId:taskId

调用TransportReindexAction执行任务

拉取任务状态

/_tasks/{reindexId}

存储Reindex任务执行状态

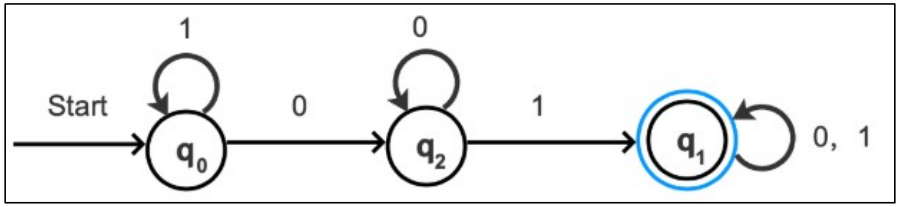
每个batch结束后更新状态

Reindex实现：sourceCluster白名单校验

确定型有穷自动机DFA

$$A = (Q, \sum, \delta, q_0, F)$$

Q：状态集合
Σ：合法的输入字符
δ：状态转移函数
δ(q1, a)=q2：从状态q1，输入字符a，转移到状态q2
q0：初始状态
F：所有拒绝状态和接受状态集合



接受所有仅在串中某个地方有01序列的0和1组成的串

TransportReindexAction::doExecute
--- 执行reindex任务：sourceCluster白名单校验

buildRemoteWhitelist构建白名单匹配自动机

为每个ip白名单构建自动机，并合(或)并这些自动机

```
Regex.simpleMatchToAutomaton(whitelist.toArray(Strings.EMPTY_ARRAY));
```

```
List<Automaton> automata = new ArrayList<>();
for (String pattern : patterns) {
    automata.add(simpleMatchToAutomaton(pattern));
}
return Operations.union(automata);
```

Regex::simpleMatchToAutomaton, 处理每个ip的wildcard字符

```
remote.whitelist: "otherhost:9200, another:9200, 127.0.10.*:9200, localhost:*"
```

```
public static Automaton simpleMatchToAutomaton(String pattern) {
    List<Automaton> automata = new ArrayList<>();
    int previous = 0;
    for (int i = pattern.indexOf('*'); i != -1; i = pattern.indexOf( ch: '*', fromIndex: i + 1)) {
        automata.add(Automata.makeString(pattern.substring(previous, i)));
        automata.add(Automata.makeAnyString());
        previous = i + 1;
    }
    automata.add(Automata.makeString(pattern.substring(previous)));
    return Operations.concatenate(automata);
}
```

串联(与)这些自动机

Reindex实现：目标索引有效性校验

源索引在远程集群，无需进行目标索引校验

判断是否满足自动创建索引规则

N

索引已存在，无需自动创建

集群设置不允许自动创建，抛异常

集群不允许自动类型检测，抛异常

索引名符合自动创建匹配规则

Y

处理别名，获取索引真实名称

若sourceIndex与targetIndex相同，抛异常

```
if (false == autoCreateIndex.shouldAutoCreate(target, clusterState)) {
    target = indexNameExpressionResolver.concreteWriteIndex(clusterState, destination).getName();
}
for (String sourceIndex : indexNameExpressionResolver.concreteIndexNames(clusterState, source)) {
    if (sourceIndex.equals(target)) {...}
}
```

TransportReindexAction::doExecute
--- 执行reindex任务：目标索引有效性校验

校验目标索引有效性（仅集群内reindex需要校验）

```
validateAgainstAliases(request.getSearchRequest(), request.getDestination(), request.getRemoteInfo(),
    indexNameExpressionResolver, autoCreateIndex, state);
```

判断是否应该自动创建target索引

如果targetIndex存在或者是一个别名，return false

```
if (resolver.hasIndexOrAlias(index, state)) {
    return false;
}
```

集群action.auto_create_index为false，且targetIndex不存在

```
if (autoCreate.autoCreateIndex == false) {
    throw new IndexNotFoundException("[" + AUTO_CREATE_INDEX_SETTING.
}
```

若index.mapper.dynamic=false，不允许自动检测mapper类型

```
if (dynamicMappingDisabled) {
    throw new IndexNotFoundException("[" + MapperService.INDEX_MAPPER_DYNAMIC_SETTING
}
```

若未配置索引匹配规则，默认行为是自动创建索引

```
if (autoCreate.expressions.isEmpty()) {
    return true;
}
```

若targetIndex符合索引匹配规则，自动创建索引

若sourceIndex == targetIndex，抛出异常

Reindex实现：任务切分

自动确定任务并发度：slices==auto

获取源索引的主分片信息

sourceIndex1：shardId[]

sourceIndex2：shardId[]

统计每个索引对应的主分片数
sourceIndex1: 2, sourceIndex2: 3

选取最小主分片数(<=20)作为任务并发度

```
Map<Index, Integer> countsByIndex = Arrays.stream(response.getGroups())
    .collect(Collectors.toMap(
    group -> group.getShardId().getIndex(),
    group -> 1,
    (sum, term) -> sum + term
));
Set<Integer> counts = new HashSet<>(countsByIndex.values());
int leastShards = Collections.min(counts);
return Math.min(leastShards, AUTO_SLICE_CEILING);
```

TransportReindexAction::doExecute
--- 执行reindex任务：任务切分

slices参数可设置reindex任务的并发度

```
BulkByScrollParallelizationHelper.startSlicedAction(
    request, bulkByScrollTask,
    ReindexAction.INSTANCE, listener, client,
    clusterService.localNode(),
    () -> {...}
);
```

Runnable

slices==auto，自动确定任务并发度

获取源索引（可多个）的主分片信息

```
ClusterSearchShardsRequest shardsRequest = new ClusterSearchShardsRequest();
shardsRequest.indices(request.getSearchRequest().indices());
client.admin().cluster().searchShards(shardsRequest, ActionListener.wrap(
    response -> {
        int actualNumSlices = countSlicesBasedOnShards(response);
        sliceConditionally(request, task, action, listener, client, node, worl
    },
```

ClusterSearchShardsResponse

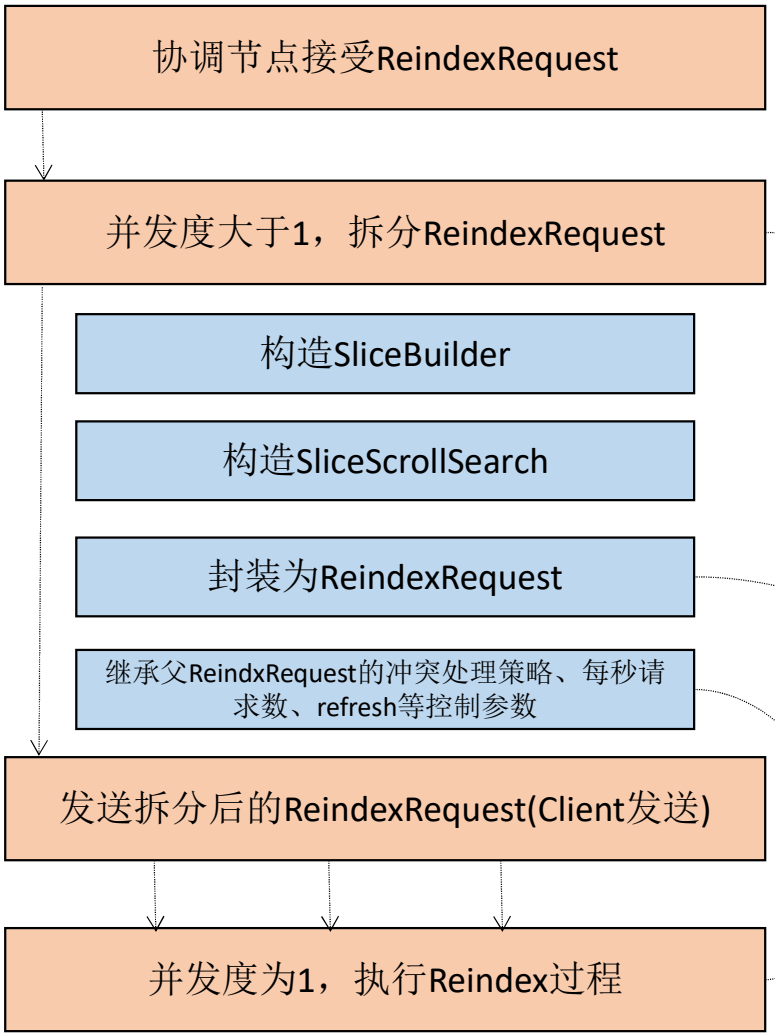
groups	ClusterSearchShardsGroup[]
nodes	DiscoveryNode[]
indicesAndFilters	Map<String, AliasFilter>

ClusterSearchShardsGroup

shardId	ShardId
shards	ShardRouting[]

统计每个源索引对应的主分片个数，选取最小值(<=20)

Reindex实现：任务切分



BulkByScrollParallelizationHelper::sliceConditionally
--- 执行reindex任务：任务切分，并执行

slices==1（只有一个主分片，或当前为拆分后的reindex请求）

获取当前查询的sliceId，设置当前task的状态

```
SliceBuilder sliceBuilder = request.getSearchRequest().source().slice();
Integer sliceId = sliceBuilder == null? null: sliceBuilder.getId();
task.setWorker(request.getRequestsPerSecond(), sliceId);
```

```
workerState = new WorkerBulkByScrollTaskState( task: this, sliceId, requestsPerSecond)
```

执行后续代码

```
workerAction.run();
```

slices>1，任务并发度大于1

标记当前task为leader task

```
leaderState = new LeaderBulkByScrollTaskState( task: this, slices);
```

按并发度拆分发送reindex请求（构造多个scrollSearch请求）

```
sendSubRequests(client, action, node.getId(), task, request, listener);
```

```
ReindexRequest(SearchRequest search, IndexRequest destination) {
    this(search, destination, setDefaults: true);
}
```

```
request.setAbortOnVersionConflict(abortOnVersionConflict).setRefresh(refresh).setTimeout(timeout)
        .setWaitForActiveShards(activeShardCount).setRetryBackoffInitialTime(retryBackoffInitialTi
        // Parent task will store result
        .setShouldStoreResult(false)
        // Split requests per second between all slices
        .setRequestsPerSecond(requestsPerSecond / totalSlices)
        // Sub requests don't have workers
        .setSlices(1);
```

subrequest的slice需要设置为1

Reindex实现：任务切分

WorkerBulkByScrollTaskState

```
private final BulkByScrollTask task;
private final Integer sliceId;
private final AtomicLong total = new AtomicLong( initialValue: 0);
private final AtomicLong updated = new AtomicLong( initialValue: 0);
private final AtomicLong created = new AtomicLong( initialValue: 0);
private final AtomicLong deleted = new AtomicLong( initialValue: 0);
private final AtomicLong noops = new AtomicLong( initialValue: 0);
private final AtomicInteger batch = new AtomicInteger( initialValue: 0);
private final AtomicLong versionConflicts = new AtomicLong( initialValue: 0);
private final AtomicLong bulkRetries = new AtomicLong( initialValue: 0);
private final AtomicLong searchRetries = new AtomicLong( initialValue: 0);
private final AtomicLong throttledNanos = new AtomicLong();
private volatile float requestsPerSecond;
```

BulkByScrollParallelizationHelper::sliceConditionally
--- 执行reindex任务：任务切分，并执行

slices==1（只有一个主分片，或当前为拆分后的reindex请求）

获取当前查询的sliceId，设置当前task的状态

```
SliceBuilder sliceBuilder = request.getSearchRequest().source().slice();
Integer sliceId = sliceBuilder == null? null: sliceBuilder.getId();
task.setWorker(request.getRequestsPerSecond(), sliceId);
```

```
workerState = new WorkerBulkByScrollTaskState( task: this, sliceId, requestsPerSecond);
```

执行后续代码

```
workerAction.run();
```

slices>1，任务并发度大于1

标记当前task为leader task

```
leaderState = new LeaderBulkByScrollTaskState( task: this, slices);
```

按并发度拆分发送reindex请求（构造多个scrollSearch请求）

```
sendSubRequests(client, action, node.getId(), task, request, listener);
```

```
for (final SearchRequest slice : sliceIntoSubRequests(request.getSearchRequest(), IdFieldMapper.NAME, totalSlices)) {
    Request requestForSlice = request.forSlice(parentTaskId, slice, totalSlices);
    ActionListener<BulkByScrollResponse> sliceListener = ActionListener.wrap(
            r -> worker.onSliceResponse(listener, slice.source().slice().getId(), r),
            e -> worker.onSliceFailure(listener, slice.source().slice().getId(), e));
    client.execute(action, requestForSlice, sliceListener);
}
```

请求拆分

发送reindex请求

Reindex实现：Search请求切分

BulkByScrollParallelizationHelper::sliceIntoSubRequests
--- 将Search请求拆分为多个，并封装为reindex请求

按并发度拆分search请求（构造多个scrollSearch请求）

```
for (final SearchRequest slice : sliceIntoSubRequests(req
```

```
static SearchRequest[] sliceIntoSubRequests(SearchRequest request, String field, int times) {
    SearchRequest[] slices = new SearchRequest[times];
    for (int slice = 0; slice < times; slice++) {
        SliceBuilder sliceBuilder = new SliceBuilder(field, slice, times);
        SearchSourceBuilder slicedSource;
        if (request.source() == null) {
            slicedSource = new SearchSourceBuilder().slice(sliceBuilder);
        } else {...}
        SearchRequest searchRequest = new SearchRequest(request);
        searchRequest.source(slicedSource);
        slices[slice] = searchRequest;
    }
    return slices;
}
```

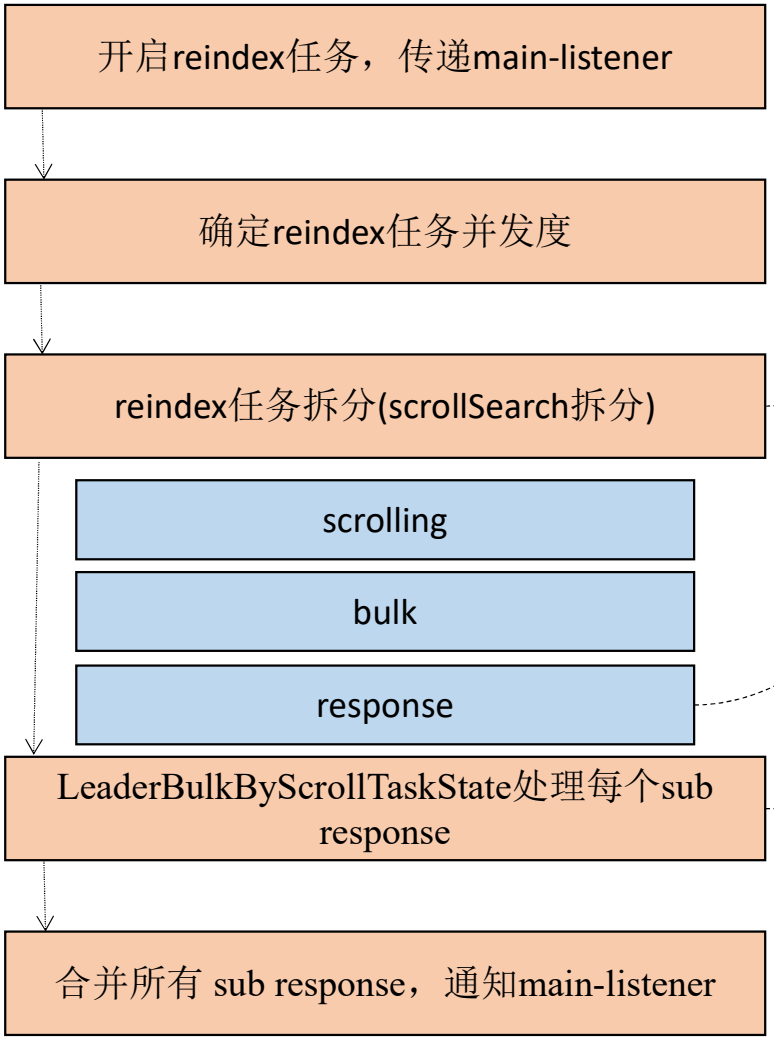
构造sliceScroll请求

将sliceScroll请求包装为ReindexRequest

```
ReindexRequest sliced = doForSlice(
    new ReindexRequest(slice, destination, setDefaults: false),
    slicingTask, totalSlices);
```

doForSlice，设置ReindexRequest的控制参数

Reindex实现：Listener与结果合并



BulkByScrollParallelizationHelper::sendSubRequests
--- 发送多个拆分后的reindex请求，注册主listener

拆分请求，注册主listener，发送请求

```
for (final SearchRequest slice :
    sliceIntoSubRequests(request.getSearchRequest(), IdFieldMapper.NAME, totalSlices)) {
    Request requestForSlice = request.forSlice(parentTaskId, slice, totalSlices);
    ActionListener<BulkByScrollResponse> sliceListener = ActionListener.wrap(
            r -> worker.onSliceResponse(listener, slice.source().slice().getId(), r),
            e -> worker.onSliceFailure(listener, slice.source().slice().getId(), e));
    client.execute(action, requestForSlice, sliceListener);
}
```

AsyncIndexBySearchAction::finishHim
--- 构造reindex response，通知listener

task记录目前reindex子任务的执行状态和进度

```
new BulkByScrollResponse(took, task.getStatus(), indexingFailures, searchFailures, timedOut);
```

通知listener

```
listener.onResponse(response);
```

LeaderBulkByScrollTaskState::onSliceResponse
--- 记录每个slice reindex结果，全部结束后发送全局response

```
results.setOnce(sliceId, new Result(sliceId, response));
```

记录当前子查询结果

```
if (runningSubtasks.decrementAndGet() != 0) {
    return;
```

判断是否所有子查询已完成

合并所有sub reindex结果，通知reindex listener

```
List<BulkByScrollResponse> responses = new ArrayList<>(results.length());
listener.onResponse(new BulkByScrollResponse(responses, task.getReasonCancelled()));
```

Reindex实现：执行Reindex任务

构造ParentTaskAssigningClient

构造AsyncIndexBySearchAction

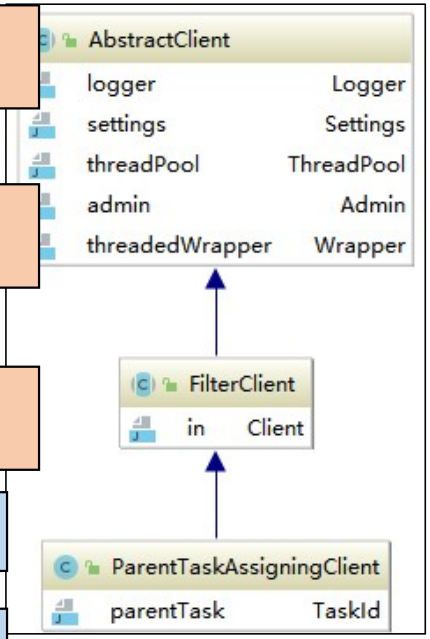
AsyncIndexBySearchAction::start

scrolling

bulk

回调
BulkIndexByScrollResponseContentListener

黄框运行于协调节点



TransportReindexAction::doExecute
--- 当slice为1时，执行reindex任务

slices==1（只有一个主分片，或当前为拆分后的reindex请求）

构造自动为所有request赋值parentTask的client

```
ParentTaskAssigningClient assigningClient = new ParentTaskAssigningClient(
    client, clusterService.localNode(), bulkByScrollTask);
```

ParentTaskAssigningClient::doExecute

为request设置parentTask，调用client发送request

```
request.setParentTask(parentTask);
super.doExecute(action, request, listener);
```

```
in().execute(action, request, listener);
```

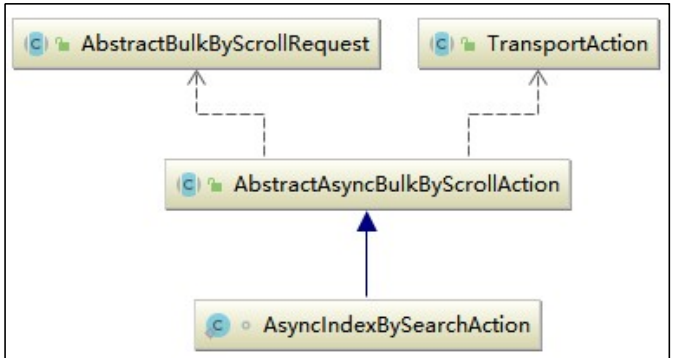
```
protected Client in() { return in; }
```

构造AsyncIndexBySearchAction

AsyncIndexBySearchAction::start，开始sub-reindex执行

```
new AsyncIndexBySearchAction(bulkByScrollTask, logger, assigningClient, threadPool, action: this, request, state,
    listener).start();
```

AsyncIndexBySearchAction是对reindex的一个简单实现，使用到了scrolling与bulk



Reindex实现：构造Client

AsyncIndexBySearchAction::construct
--- 根据源集群构造client

集群内reindex，直接使用NodeClient

集群间reindex，构造RestClient

```
if (mainRequest.getRemoteInfo() != null) {
    RemoteInfo remoteInfo = mainRequest.getRemoteInfo();
    createdThreads = synchronizedList(new ArrayList<>());
    RestClient restClient = buildRestClient(remoteInfo, mainActio
    return new RemoteScrollableHitSource(logger, backoffPolicy, t
```

构造rest search请求

```
StringBuilder path = new StringBuilder("/");
addIndexesOrTypes(path,  name: "Index", searchRequest.indices());
addIndexesOrTypes(path,  name: "Type", searchRequest.types());
path.append("_search");
Request request = new Request( method: "POST", path.toString());
```

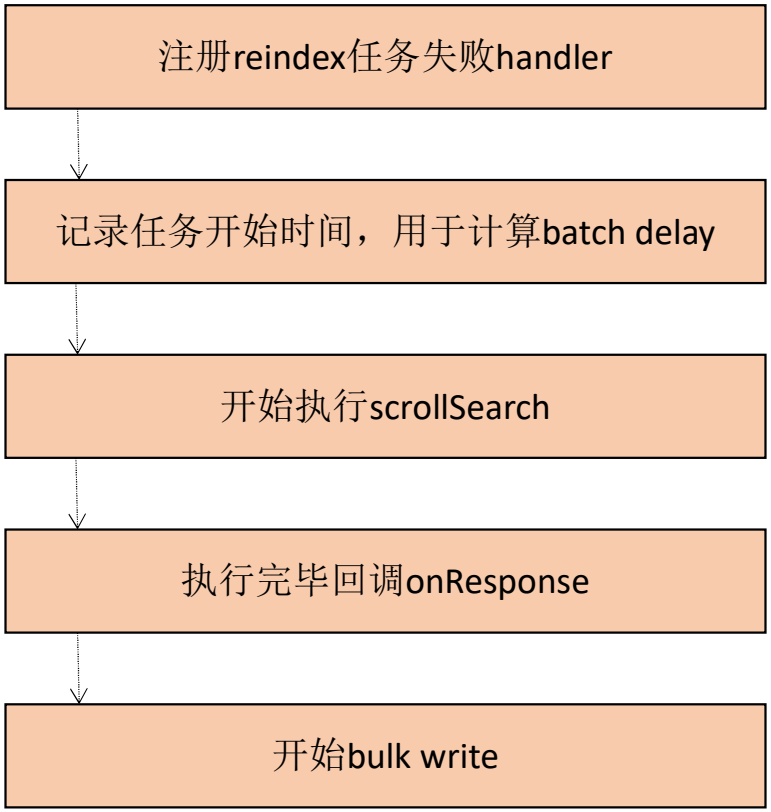
处理scroll，统一时间单位，兼容低版本

```
TimeValue keepAlive = searchRequest.scroll().keepAlive();
// V_5_0_0
if (remoteVersion.before(Version.fromId(5000099))) {
    /* Versions of Elasticsearch before 5.0 couldn't parse nanos or micros
     * so we toss out that resolution, rounding up because more scroll
     * timeout seems safer than less. */
    keepAlive = timeValueMillis((long) Math.ceil(keepAlive.millisFrac()));
}
request.addParameter( name: "scroll", keepAlive.getStringRep());
```

处理source和query等参数，json序列化

使用RemoteResponseParse解析Rest请求结果

Reindex实现：执行Reindex任务



AsyncIndexBySearchAction::start
--- 执行reindex任务

注册reindex失败处理handler

任务取消

```
if (task.isCancelled()) {
    logger.debug( message: "[{}]: finishing early
    finishHim( failure: null);
    return;
```

scrollSearch异常

```
} catch (Exception e) {
    finishHim(e);
```

构造response并回调

```
if (failure == null) {
    BulkByScrollResponse response = buildResponse(
            timeValueNanos(System.nanoTime() - startTime.get()),
            indexingFailures, searchFailures, timedOut);
    listener.onResponse(response);
} else {
    listener.onFailure(failure);
}
```

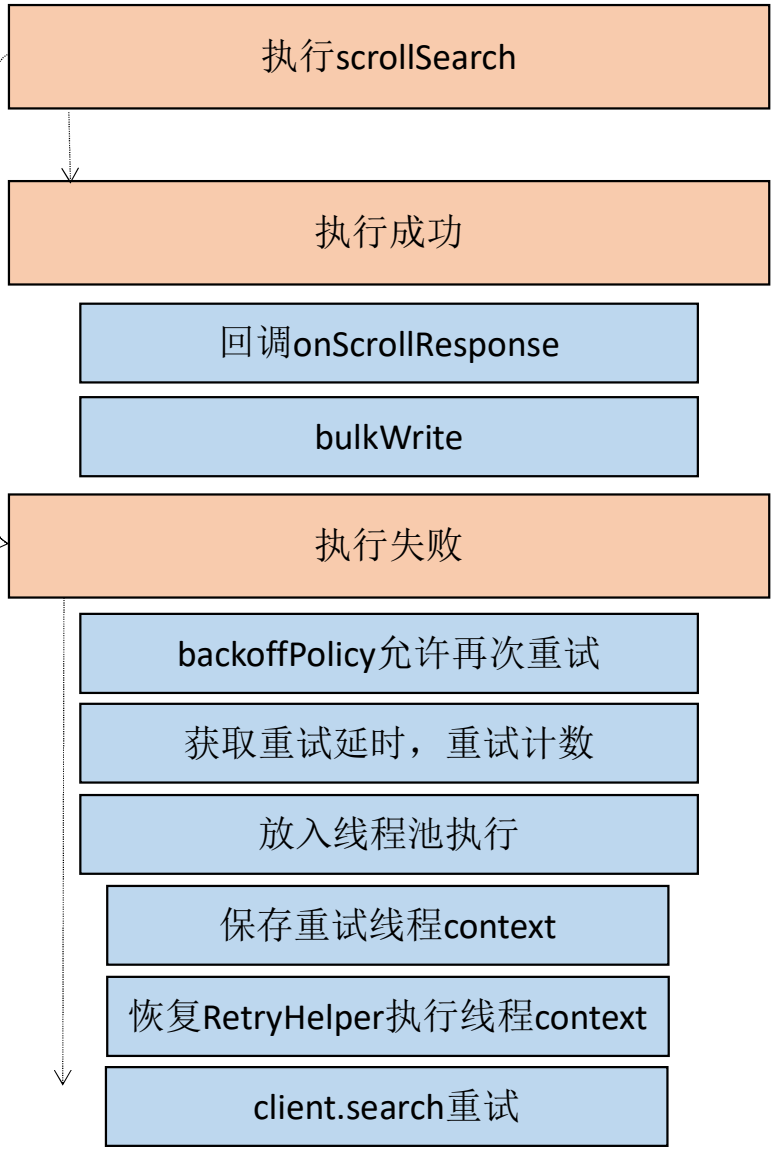
ClientScrollableHitSource::start，开始执行scrollSearch

```
startTime.set(System.nanoTime());
scrollSource.start(response -> onScrollResponse(timeValueNanos(System.nanoTime()), lastBatchSize: 0, response));
```

执行完毕，ClientScrollableHitSource回调onScrollResponse

处理scrollSearch结果，构造并发送bulkRequest

Reindex实现：Scrolling



ClientScrollableHitSource::start
--- 执行scrollSearch

doStart，设置出错重试

```
searchWithRetry(
    listener -> client.search(firstSearchRequest, listener),
    r -> consume(r, onResponse));
```

通过client发送SearchRequest，执行scrollSearch

执行回调AsyncIndexBySearchAction::onScrollResponse

searchWithRetry，带重试机制的scrollSearch

构造RetryHelper

```
RetryHelper helper = new RetryHelper();
```

为RetryHelper添加上下文管理功能

```
// Wrap the helper in a runnable that preserves
// the current context so we keep it on retry.
helper.retryWithContext = threadPool.getThreadContext().preserveContext(helper);
```

执行RetryHelper，执行scrollSearch

```
helper.run();
```

RetryHelper

backoffPolicy允许再次重试

获取重试延时，重试次数计数

放入线程池执行

Reindex实现：Scrolling

RetryHelper::run
--- 执行scrollSearch

```
private void searchWithRetry(Consumer<ActionListener<SearchResponse>> action, Consumer<SearchResponse> onResponse) {

    class RetryHelper extends AbstractRunnable implements ActionListener<SearchResponse> {
        private final Iterator<TimeValue> retries = backoffPolicy.iterator();
        /**
         * The runnable to run that retries in the same context as the original call.
         */
        private Runnable retryWithContext;
        private volatile int retryCount = 0;

        @Override
        protected void doRun() throws Exception {
            action.accept( t: this);
        }

        @Override
        public void onResponse(SearchResponse response) { onResponse.accept(response); }

        @Override
        public void onFailure(Exception e) {
            if (ExceptionsHelper.unwrap(e, EsRejectedExecutionException.class) != null) {
                if (retries.hasNext()) {
                    retryCount += 1;
                    TimeValue delay = retries.next();
                    logger.trace(() -> new ParameterizedMessage( messagePattern: "retrying rejected search after [{}]", delay), e);
                    countSearchRetry.run();
                    threadPool.schedule(retryWithContext, delay, ThreadPool.Names.SAME);
                } else {
                    logger.warn(() -> new ParameterizedMessage(
                            messagePattern: "giving up on search because we retried [{}] times without success", retryCount), e);
                    fail.accept(e);
                }
            } else {
                logger.warn( message: "giving up on search because it failed with a non-retryable exception", e);
                fail.accept(e);
            }
        }
    }
    RetryHelper helper = new RetryHelper();
    // Wrap the helper in a runnable that preserves the current context so we keep it on
    helper.retryWithContext = threadPool.getThreadContext().preserveContext(helper);
    helper.run();
}
```

使RetryHelper具备上下文切换能力

run

```
threadsOriginalContext = stashContext();
try {
    creatorsContext.restore();
    whileRunning = true;
    in.doRun();
```

保存执行线程上下文

恢复origin call上下文

doRun

onResponse

onFailure

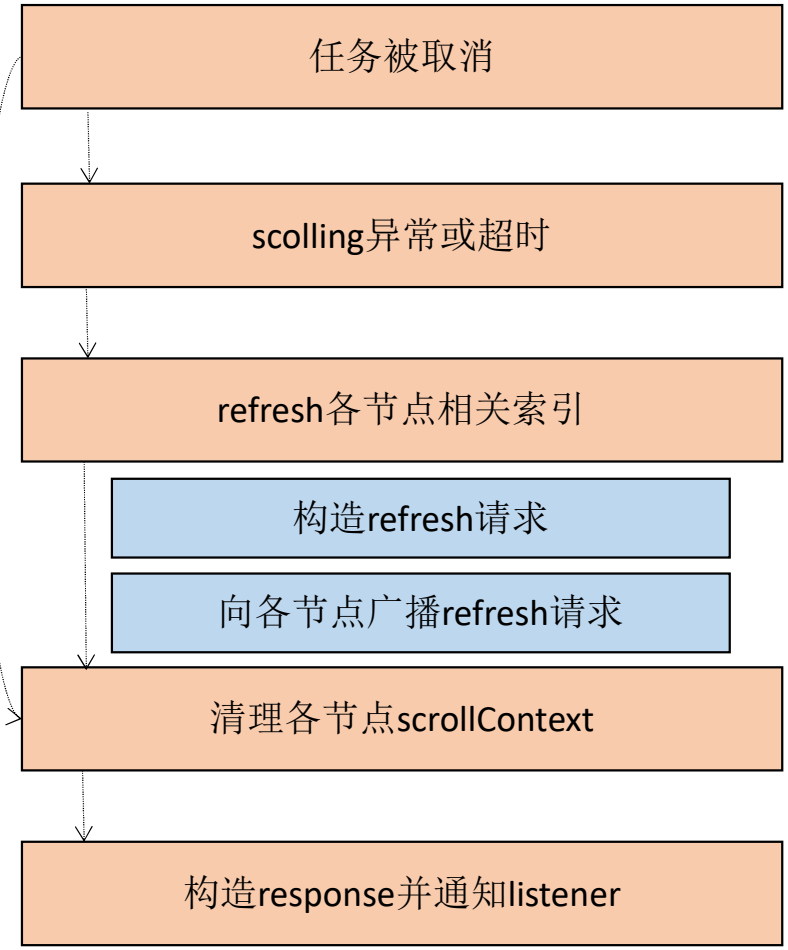
onAfter

```
public void onAfter() {
    try {
        in.onAfter()
    } finally {
        if (threadsOriginalContext != null) {
            threadsOriginalContext.restore();
        }
    }
}
```

RetryHelper::onAfter

恢复线程原有context

Reindex实现：处理scoll response



AsyncIndexBySearchAction::onScrollResponse
--- 处理scrollSearch结果

判断任务是否被取消

```
if (task.isCancelled()) {
    logger.debug( message: "[{}]: fi
    finishHim( failure: null);
    return;
}
```

scrolling存在异常或超时

refresh已经写入的数据

```
if ((response.getFailures().size() > 0)|| response.isTimedOut()) {
    refreshAndFinish(emptyList(), response.getFailures(), response.isTimedOut());
    return;
}
```

```
RefreshRequest refresh = new RefreshRequest();
refresh.indices(destinationIndices.toArray(new String[destinat
logger.debug( message: "[{}]: refreshing", task.getId());
client.admin().indices().refresh(refresh, new ActionListener<R
```

停止任务

```
public void onResponse(RefreshResponse response) {
    finishHim( failure: null, indexingFailures, searchFailures, timedOut);
}
```

清理各节点的scrollContext

```
scrollSource.close(() -> {
    if (failure == null) {
```

Reindex实现：构造Bulk延时任务

构造bulk任务

计算任务延时

计算上一个batch的理论耗时
(scrolling+bulk总耗时)

P=lastBatchSIze / requestsPerSecond

下一个batch的最早开始时间
earliestStartTIme = scroll开始时间 + P

waitTime=earliestStartTIme - now

开启延时任务(DelayedPrepareBulkRequest)

保存延时任务,用于rethrottle调整reindex速率

AsyncIndexBySearchAction::onScrollResponse
--- 处理scrollSearch结果，构造bulk任务，并延时执行

任务异常处理

构造bulk任务

```
AbstractRunnable prepareBulkRequestRunnable = new AbstractRunnable() {
    @Override
    protected void doRun() throws Exception {
        prepareBulkRequest(timeValueNanos(System.nanoTime()), response);
    }
    @Override
    public void onFailure(Exception e) { finishHim(e); }
};
```

用于计算batch delay

清理scrollContext，结束任务，回调listener

赋予bulk任务线程上下文管理能力

```
prepareBulkRequestRunnable = (AbstractRunnable)
    threadPool.getThreadContext().preserveContext(prepareBulkRequestRunnable);
```

构建延时任务

```
worker.delayPrepareBulkRequest(threadPool, lastBatchStartTime, lastBatchSize, prepareBulkRequestRunnable);
```

WorkerBulkByScrollTaskState::delayPrepareBulkRequest
--- 计算任务延时，构造延时任务并保存

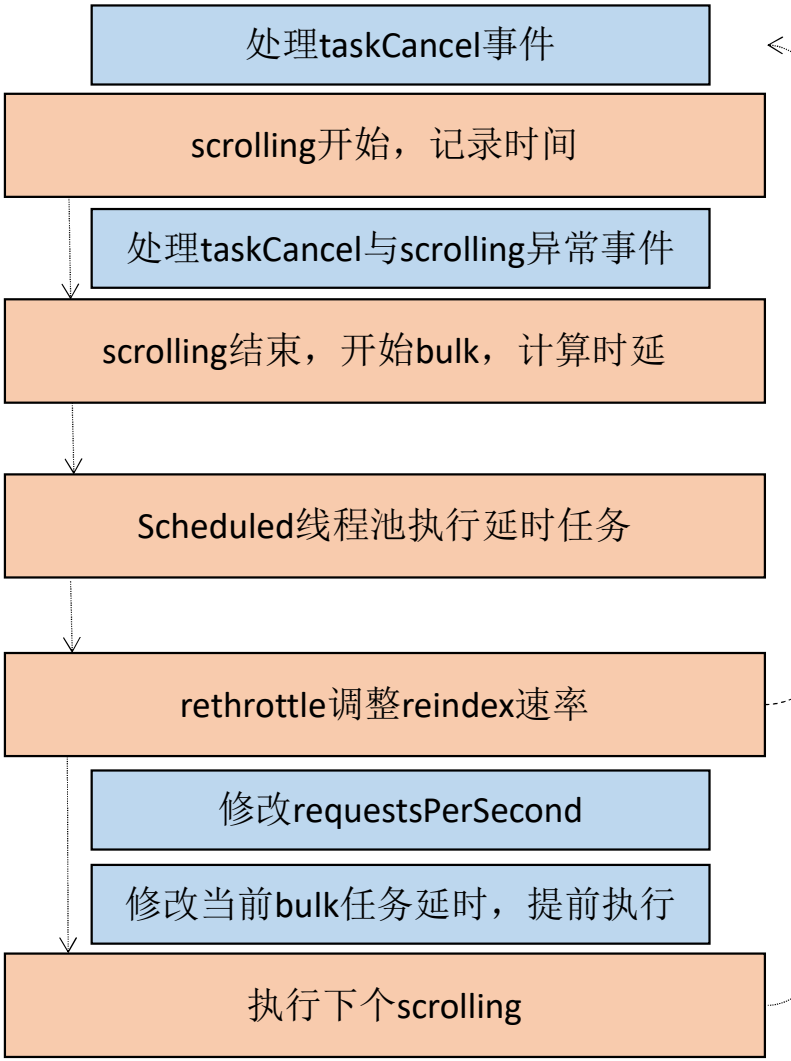
计算任务延时

```
long earliestNextBatchStartTime = now.nanos() + (long) perfectlyThrottledBatchTime(lastBatchSize);
long waitTime = min(MAX_THROTTLE_WAIT_TIME.nanos(), max(0, earliestNextBatchStartTime - System.nanoTime()));
```

开启延时任务并保存

```
delayedPrepareBulkRequestReference.set(new DelayedPrepareBulkRequest(threadPool, getRequestsPerSecond(),
    delay, new RunOnce(prepareBulkRequestRunnable)));
```

Reindex实现：延时任务细节与rethrottle



一个reindex循环

DelayedPrepareBulkRequest::construct
--- 执行reindex任务与调控reindex速率

执行延时任务

```
this.scheduled = threadPool.schedule(() -> {
    throttledNanos.addAndGet(delay.nanos());
    command.run();
}, delay, ThreadPool.Names.GENERIC);
```

DelayedPrepareBulkRequest::rethrottle
--- 调控reindex速率

不允许降低reindex速度,可能导致SearchContext过期(默认5min)

```
if (newRequestsPerSecond < requestsPerSecond) {
    logger.debug( message: "[{}]: skipping rescheduling beca
        newRequestsPerSecond, requestsPerSecond);
    return this;
```

获取当前运行任务的剩余时延(<0表示任务已被执行,不做处理)

```
long remainingDelay = scheduled.getDelay(TimeUnit.NANOSECONDS);
```

重新计算延时，并另起一个相同的延时任务

```
timeValueNanos(round(remainingDelay * requestsPerSecond / newRequestsPerSecond));
logger.debug( message: "[{}]: rescheduling for [{}] in the future", task.getId(), newDel
return new DelayedPrepareBulkRequest(threadPool, requestsPerSecond, newDelay, command);
```

任务代理RunOnce保障相同任务只执行一次（调整后的任务）

```
public void run() {
    if (hasRun.compareAndSet(false, true)) {
        delegate.run();
    }
}
```

Reindex实现：Bulk Write

处理任务取消事件

↓

scrollSearch无结果则结束此reindex

↓

根据scrolling结果构造bulkRequest

↓

使用client发送bulk请求

↓

回调：统计bulk结果，执行下一轮reindex

AsyncIndexBySearchAction::prepareBulkRequest
--- 创建bulk请求并发送

处理任务取消事件

scrolling无结果则结束reindex

```
if (response.getHits().isEmpty()) {
    refreshAndFinish(emptyList(), emptyList(), timedOut: false);
```

根据每个batch的size需要，截短scrolling结果

```
long remaining = max(0, mainRequest.getSize() - worker.getSuccessfullyProcessed());
if (remaining < hits.size()) {
    hits = hits.subList(0, (int) remaining);
```

构造bulkRequest

```
BulkRequest bulkRequest = new BulkRequest();
for (ScrollableHitSource.Hit doc : docs) {
    if (accept(doc)) {
        RequestWrapper<?> request = scriptApplier.apply(copyMetadata(buildRequest(doc), doc), doc);
        if (request != null) {
            bulkRequest.add(request.self());
        }
    }
}
return bulkRequest;
```

拒绝没有存储_source的文档

构造bulkRequest

发送bulk请求

```
bulkRetry.withBackoff(client::bulk, request, new ActionListener<BulkResponse>() {
    @Override
    public void onResponse(BulkResponse response) { onBulkResponse(thisBatchStartTim
```

bulk回调：bulk结果计数，开始下一轮reindex(scrolling+bulk)

Reindex实现：BulkRequest构造细节

BulkRequest

IndexRequest -> doc1
[version_type, version, index, type, source, routing, pipeline]

IndexRequest -> doc2
[version_type, version, index, type, source, routing, pipeline]

IndexRequest -> doc3
[version_type, version, index, type, source, routing, pipeline]

Versioning

Each bulk item can include the version value using the `version` field. It automatically follows the behavior of the index / delete operation based on the `_version` mapping. It also support the `version_type` (see versioning).

reindex的routing保留策略

```
if (routingSpec.startsWith("=")) {
    super.copyRouting(request, mainRequest.getDestination().routing().substring(1));
    return;
}
switch (routingSpec) {
case "keep":
    super.copyRouting(request, routing);
    break;
case "discard":
    super.copyRouting(request, routing: null);
    break;
```

=text，设置为text

设置为与文档routing相同

抛弃文档routing

AsyncIndexBySearchAction::buildRequest
--- 创建bulk请求

构造IndexRequest，处理dest和type

```
IndexRequest index = new IndexRequest();
index.index(mainRequest.getDestination().index());
if (mainRequest.getDestination().type() == null) {
    index.type(doc.getType());
} else {
    index.type(mainRequest.getDestination().type());
}
```

设置IndexRequest的版本控制策略

INTERNAL，索引全部文档或覆盖相同Id文档

IndexRequest.version设置为MATCH_ANY(-3)或MATCH_DELETED(-4)

```
if (index.versionType() == INTERNAL) {
    assert doc.getVersion() == -1 : "fetched version when
    index.version(mainRequest.getDestination().version());
```

EXTERNAL[_GT][_GTE],根据文档版本决定是否索引

IndexRequest.version设置为外部文档版本

```
index.version(doc.getVersion());
```

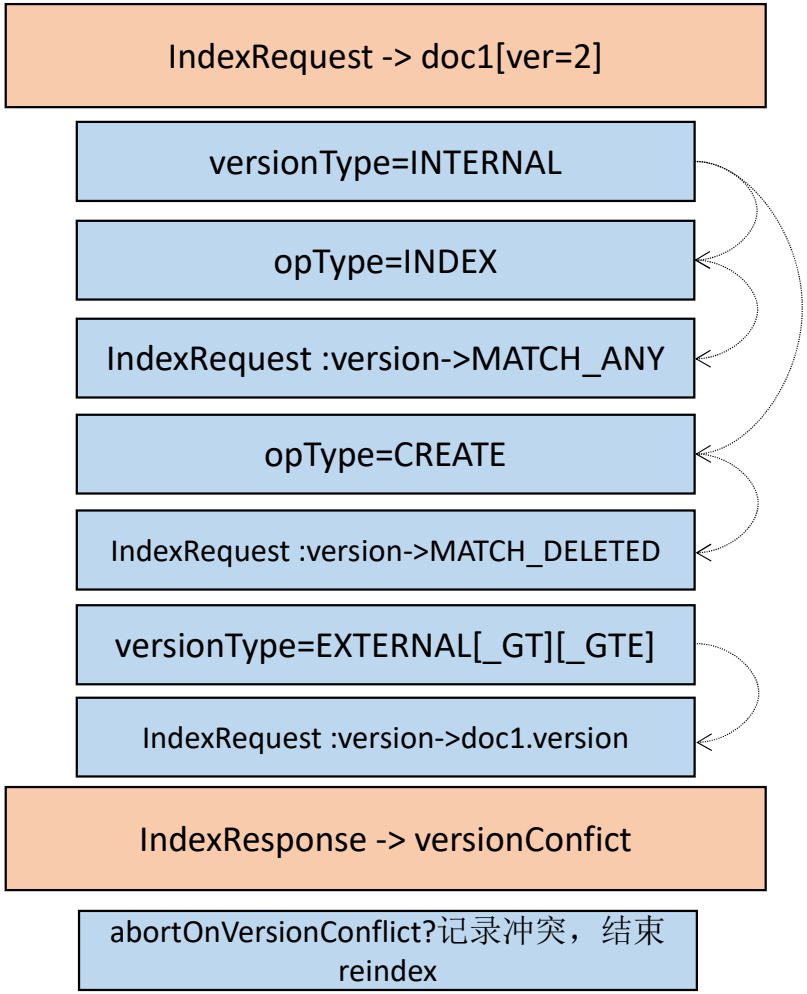
设置IndexRequest的source和source xcontent type

```
index.source(doc.getSource(), doc.getXContentType());
```

设置routing和pipline

```
index.routing(mainRequest.getDestination().routing());
index.setPipeline(mainRequest.getDestination().getPipeline());
```

Reindex实现：文档版本控制



conflicts

(Optional, enum) Set to `proceed` to continue reindexing even if there are conflicts. Defaults to `abort`.

AsyncIndexBySearchAction::buildRequest
--- 创建bulk请求

设置IndexRequest的版本控制策略

versionType = INTERNAL 或 EXTERNAL[_GT][_GTE]

versionType = INTERNAL，索引全部文档或覆盖相同Id文档

opType=INDEX，索引全部文档，包括已存在的

opType=CREATE，只索引不存在文档，否则产生一个文档冲突

```
if (index.versionType() == INTERNAL) {
    assert doc.getVersion() == -1 : "fetched version when we didn't have to";
    index.version(mainRequest.getDestination().version());
```

通过设置IndexRequest的version来控制索引策略

索引过程无视文档版本

只索引不存在或已删除文档

versionType = EXTERNAL，索引过程需考虑文档版本

直接设置IndexRequest的version为文档版本号

```
index.version(doc.getVersion());
```

Reindex实现：Bulk回调

遍历每个文档索引Response

记录文档索引异常信息

版本冲突计数

CREATE、UPDATE、DELETE计数

若任务已取消，清除scrollContext，结束reindex

存在文档索引异常，refresh后结束reindex

已索引足够数量文档，refresh后结束reindex

开始下一轮scrolling
重置scrolling上下文超时时间，防止回收

在第一次scrolling的keepAlive时间基础上，加上上一轮reindex等待时间

默认scrolling为5分钟超时

AsyncIndexBySearchAction::onBulkResponse
--- 处理bulkResponse，开启下一次scrolling与bulk

依次处理每个IndexRequest对应的Response

文档Index失败，记录失败原因

若原因为版本冲突，reindexState记录冲突数加一

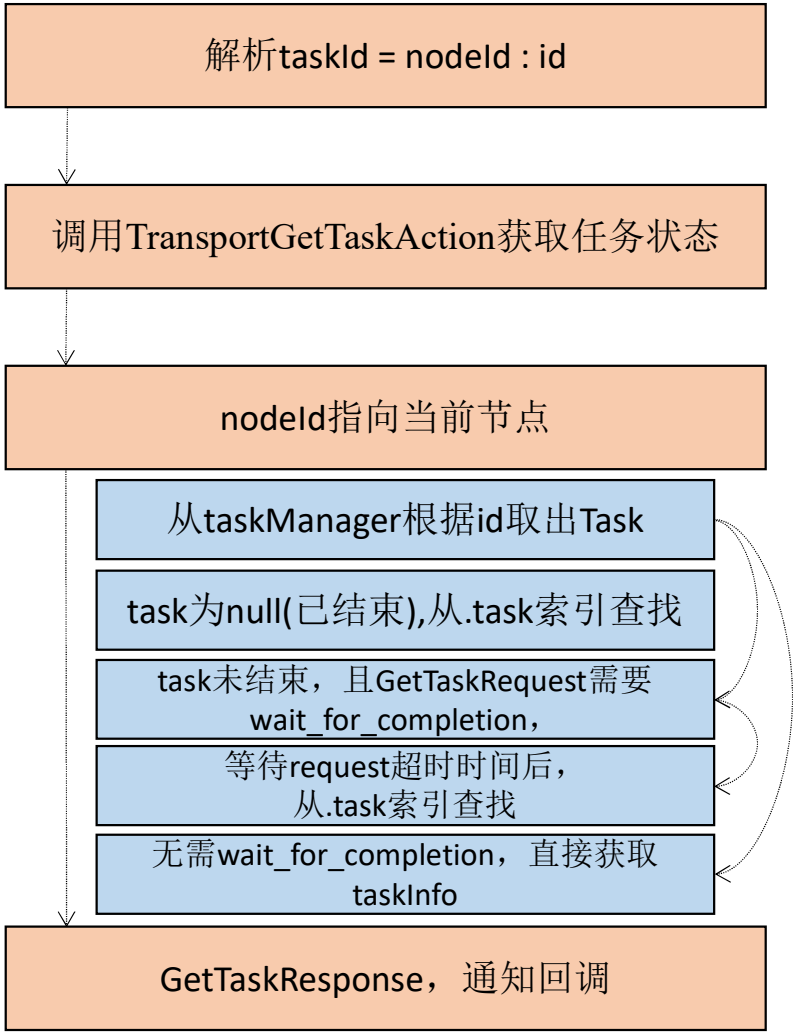
```
if (failure.getStatus() == CONFLICT) {
    worker.countVersionConflict();
    if (false == mainRequest.isAbortOnVersionConflict())
        return;
    }
}
failures.add(failure);
```

根据文档索引类型(CREATE、INDEX、UPDATE、DELETE)，将reindexState对应计数加一

```
switch (item.getOpType()) {
    case CREATE:
    case INDEX:
        if (item.getResponse().getResult() == DocWriteResponse.Result.CREATED) {
            worker.countCreated();
        } else {
            worker.countUpdated();
        }
        break;
    case UPDATE:
        worker.countUpdated();
        break;
    case DELETE:
        worker.countDeleted();
        break;
}
```

任务取消、bulk有文档索引异常、已索引要求文档数，refresh之后结束reindex

Task管理：获取任务状态



RestGetTaskAction::prepareRequest
--- 构造GetTaskRequest，使用NodeClient处理

处理get task参数，构造request

解析taskId = nodeId：id

```
TaskId taskId = new TaskId(request.param( key: "task_id"));
boolean waitForCompletion = request.paramAsBoolean( key: "wait_for_completion",  defaultValue: false);
TimeValue timeout = request.paramAsTime( key: "timeout",  defaultValue: null);

GetTaskRequest getTaskRequest = new GetTaskRequest();
getTaskRequest.setTaskId(taskId);
getTaskRequest.setWaitForCompletion(waitForCompletion);
getTaskRequest.setTimeout(timeout);
```

使用NodeClient处理GetTaskRequest

```
channel -> client.admin().cluster().getTask(getTaskRequest, new RestToXContentListener<>(channel));
```

ClusterAdminClient::getTask
--- 调用TransportGetTaskAction::execute

task创建节点为当前节点

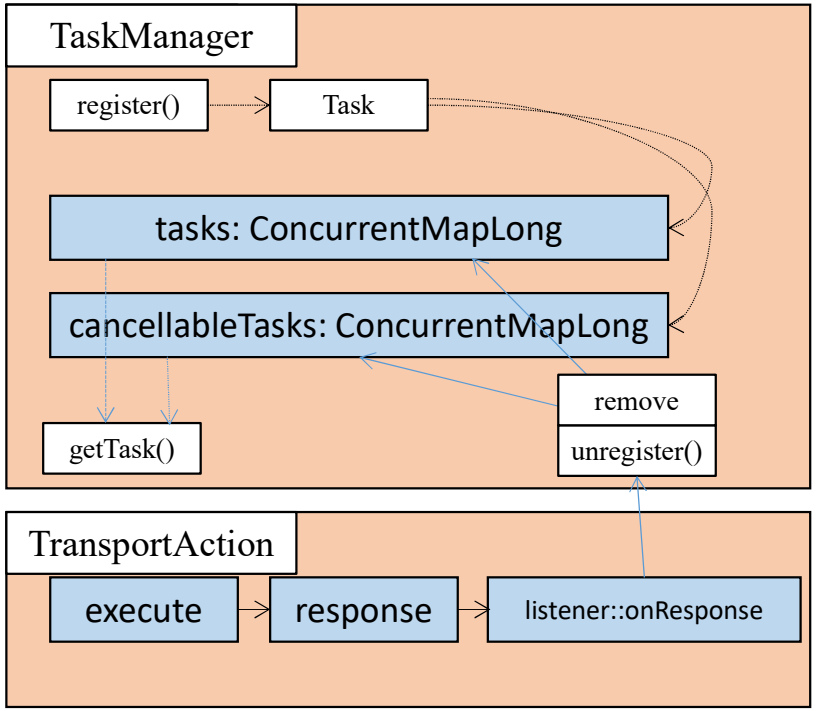
从taskManager取出task以及taskInfo,通知回调

task创建节点为远程节点

通过transportService rpc调用对应节点的GetTaskAction

获取taskInfo，通知回调

Task管理：任务注册与获取



TaskManager::register
--- 构造并记录Task, 只记录运行中的任务

调用request::createTask创建请求相关任务

根据任务是否可取消，存放于不同的map中

```
if (task instanceof CancellableTask) {
    registerCancellableTask(task);
} else {
    Task previousTask = tasks.put(task.getId(), task);
    assert previousTask == null;
}
return task;
```

调用request::getTask根据Id获取任务

```
public Task getTask(long id) {
    Task task = tasks.get(id);
    if (task != null) {
        return task;
    } else {
        return getCancellableTask(id);
    }
}
```

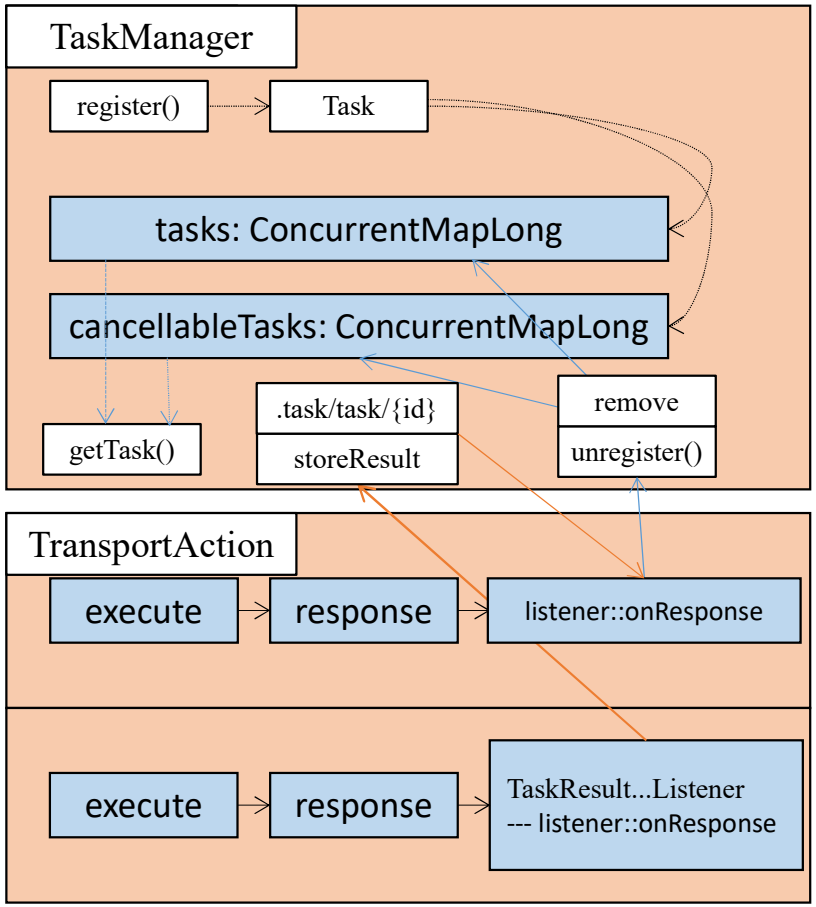
TransportAction::execute::listener::onResponse, 接收到结果后

taskManager::unregister

通知action回调

```
public Task unregister(Task task) {
    logger.trace( message: "unregister task for id: {}", task.getId());
    if (task instanceof CancellableTask) {
        CancellableTaskHolder holder = cancellableTasks.remove(task.getId());
        if (holder != null) {
            holder.finish();
            return holder.getTask();
        } else {
            return null;
        }
    } else {
        return tasks.remove(task.getId());
```

Task管理：任务持久化



TaskManager::storeResult
--- 将任务信息持久化到.task/task/{taskId}中

构造TaskResult(task, localNode, response)

TaskResultService::storeResult，将TaskResult记录至索引中

CreateIndexRequest，创建索引

```
client.admin().indices().create(createIndexRequest,
```

创建文档索引请求，并使用TaskResult填充请求source

```
IndexRequestBuilder index = client.prepareIndex(TASK_INDEX, TASK_TYPE, taskResult.getTask().getTaskId().toString());
try (XContentBuilder builder = XContentFactory.contentBuilder(Requests.INDEX_CONTENT_TYPE)) {
    taskResult.toXContent(builder, ToXContent.EMPTY_PARAMS);
    index.setSource(builder);
```

持久化TaskResult

TransportAction::execute
--- 根据request设置决定是否持久化task

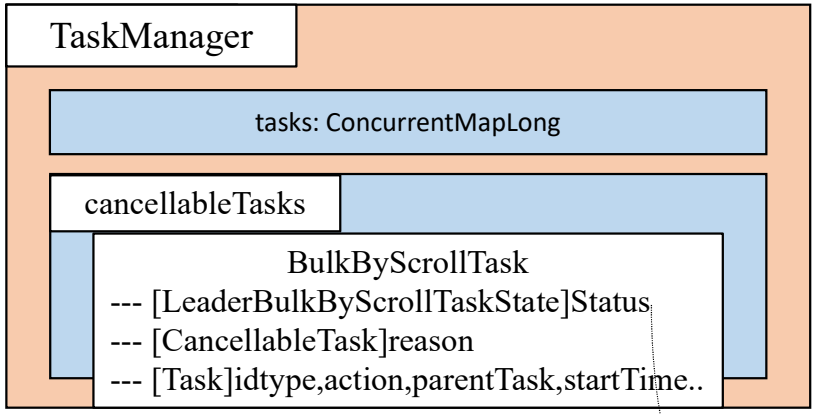
若request需要持久化task，包装listener，增加持久化功能

```
if (task != null && request.getShouldStoreResult()) {
    listener = new TaskResultStoringActionListener<>(taskManager, task, listener);
}
```

TaskResultStoringActionListener::onResponse ::onFailure

```
try {
    taskManager.storeResult(task, response, delegate);
} catch (Exception e) {
    delegate.onFailure(e);
}
```

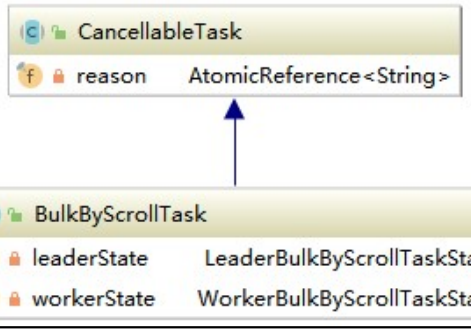
Task管理：Reindex任务



ReindexRequest::createTask
--- 创建BulkByScrollTask，记录reindex进度和状态

TaskManager::register调用ReindexRequest::createTask

```
public Task createTask(long id, String type, String action, TaskId parentTaskId, Map<String,
    return new BulkByScrollTask(id, type, action, getDescription(), parentTaskId, headers);
}
```



BulkByScrollTask::taskInfo，获取任务状态信息

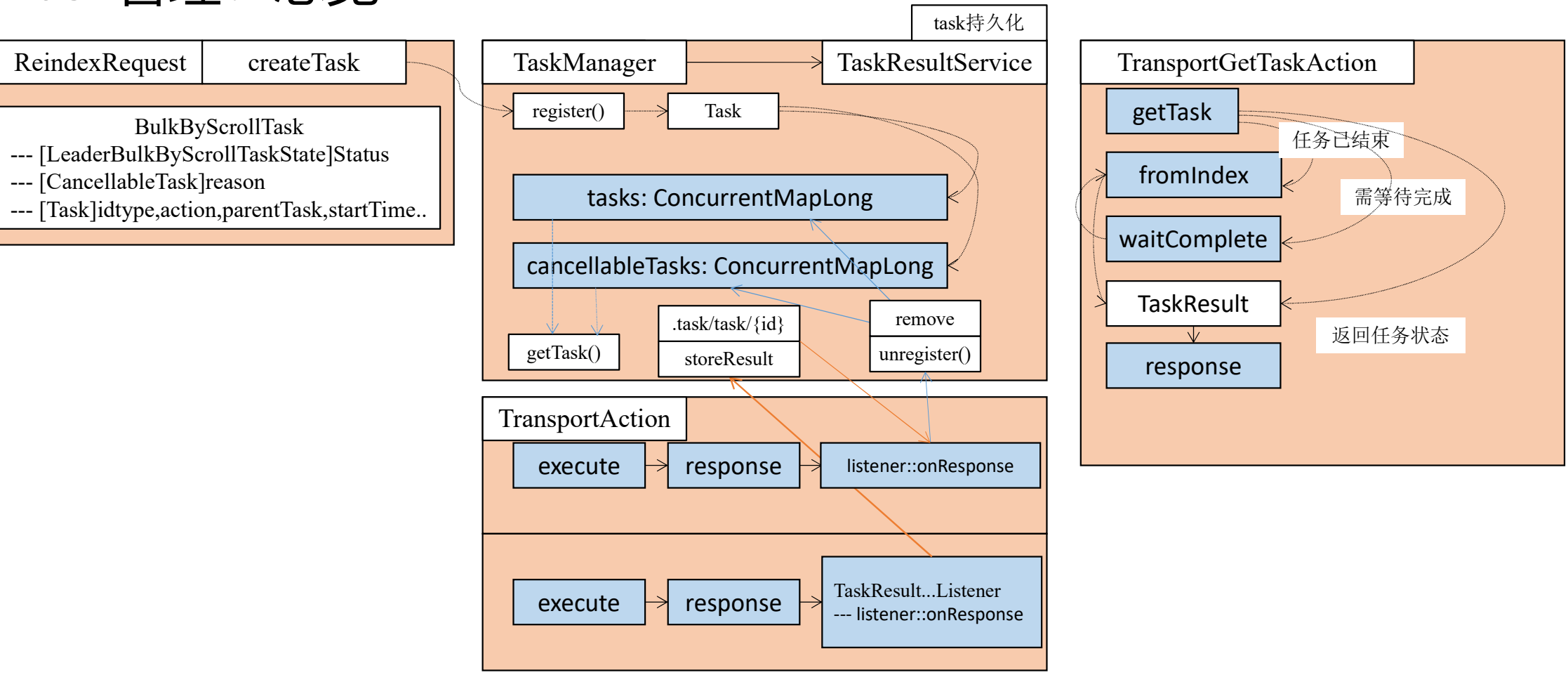
```
if (detailed) {
    description = getDescription();
    status = getStatus();
}
return taskInfo(localNodeId, description, status);
```

BulkByScrollTask::LeaderBulkByScrollTaskState::getStatus

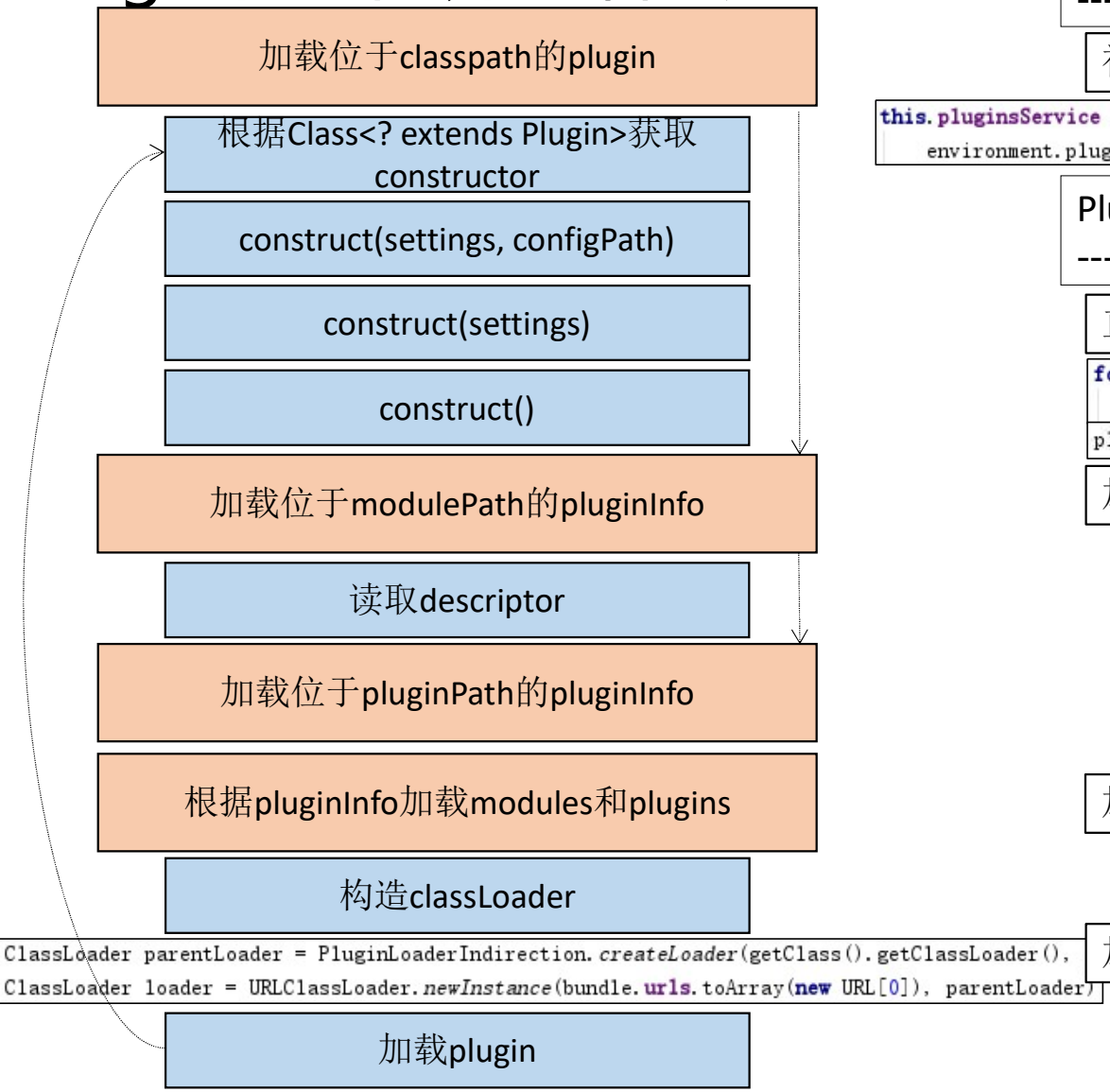
BulkByScrollTask::Status存储reindex任务进度

```
public Status(Integer sliceId, long total, long updated, long created, long deleted, int
        long bulkRetries, long searchRetries, TimeValue throttled, float requestsPerSeco
        TimeValue throttledUntil) {
    this.sliceId = sliceId == null ? null : checkPositive(sliceId, name: "sliceId");
    this.total = checkPositive(total, name: "total");
    this.updated = checkPositive(updated, name: "updated");
    this.created = checkPositive(created, name: "created");
    this.deleted = checkPositive(deleted, name: "deleted");
    this.batches = checkPositive(batches, name: "batches");
    this.versionConflicts = checkPositive(versionConflicts, name: "versionConflicts");
    this.noops = checkPositive(noops, name: "noops");
    this.bulkRetries = checkPositive(bulkRetries, name: "bulkRetries");
    this.searchRetries = checkPositive(searchRetries, name: "searchRetries");
    this.throttled = throttled;
    this.requestsPerSecond = requestsPerSecond;
    this.reasonCancelled = reasonCancelled;
    this.throttledUntil = throttledUntil;
    this.sliceStatuses = emptyList();
}
```

Task管理：总览


task持久化
ReindexRequest
createTask
BulkByScrollTask
--- [LeaderBulkByScrollTaskState]Status
--- [CancellableTask]reason
--- [Task]idtype,action,parentTask,startTime..
TaskManager
TaskResultService
register()
Task
tasks: ConcurrentMapLong
cancellableTasks: ConcurrentMapLong
getTask()
.task/task/{id}
storeResult
remove
unregister()
TransportAction
execute
response
listener::onResponse
execute
response
TaskResult...Listener
--- listener::onResponse
TransportGetTaskAction
getTask
任务已结束
fromIndex
需等待完成
waitComplete
TaskResult
返回任务状态
response

Plugin管理简介：插件加载



Node::construct
--- 初始化PluginService，加载plugin，应用plugin

初始化pluginService

```
this.pluginsService = new PluginsService(tmpSettings, environment.configFile(), environment.modulesFile(),
    environment.pluginsFile(), classpathPlugins);
```

PluginService::construct
--- 加载plugin，classpathPlugin -> moduleDirectory -> pluginDirectory

直接加载位于classpath的plugin

```
for (Class<? extends Plugin> pluginClass : classpathPlugins) {
    Plugin plugin = loadPlugin(pluginClass, settings, configPath);
pluginsLoaded.add(new Tuple<>(pluginInfo, plugin));
```

加载位于moduleDirectory的pluginInfo

读取plugin descriptor，获取plugin信息

```
Set<Bundle> modules = getModuleBundles(modulesDirectory);
for (final Path plugin : findPluginDirs(directory)) {
    final Bundle bundle = readPluginBundle(bundles, plugin, type);
info = PluginInfo.readFromProperties(plugin);Bundle bundle = new Bundle(info, plugin
seenBundles.addAll(modules);
```

加载位于pluginDirectory的pluginInfo

```
Set<Bundle> plugins = getPluginBundles(pluginsDirectory);
seenBundles.addAll(plugins);
```

加载module与plugin

```
List<Tuple<PluginInfo, Plugin>> loaded = loadBundles(seenBundles);
pluginsLoaded.addAll(loaded);
```

Plugin管理简介：ActionPlugin应用

Node::construct
--- 初始化PluginService，加载plugin，应用plugin

pluginService过滤出所有ActionPlugin

ActionModule应用所有ActionPlugin

```
ActionModule actionModule = new ActionModule( transportClient: false, settings, clusterModule.getIndexNameExpressionResolver(),
    settingsModule.getIndexScopedSettings(), settingsModule.getClusterSettings(), settingsModule.getSettingsFilter(),
    threadPool, pluginsService.filterPlugins(ActionPlugin.class), client, circuitBreakerService, usageService);
modules.add(actionModule);
```

注册插件的TransportAction

注册插件的RestHandler

注册插件的ActionFilter

ActionModule::construct
--- 初始化PluginService，加载plugin，应用plugin

注册插件的transportAction

```
actionPlugins.stream().flatMap(p -> p.getActions().stream()).forEach(actions::register);
```

注册插件的restHandler，使之能处理http请求

```
for (ActionPlugin plugin : actionPlugins) {
    for (RestHandler handler : plugin.getRestHandlers(settings, restController, clusterSettings, indexScopedSettings,
            settingsFilter, indexNameExpressionResolver, nodesInCluster)) {
        registerHandler.accept(handler);
    }
}
```

注册插件的actionFilter

```
return new ActionFilters(
    Collections.unmodifiableSet(
        actionPlugins.stream().flatMap(p -> p.getActionFilters().stream()).
            collect(Collectors.toSet())));
```

ReindexPlugin

```java
public class ReindexPlugin extends Plugin implements ActionPlugin {
    public static final String NAME = "reindex";

    @Override
    public List<ActionHandler<? extends ActionRequest, ? extends ActionResponse>> getActions() {
        return Arrays.asList(new ActionHandler<>(ReindexAction.INSTANCE, TransportReindexAction.class),
                new ActionHandler<>(UpdateByQueryAction.INSTANCE, TransportUpdateByQueryAction.class),
                new ActionHandler<>(DeleteByQueryAction.INSTANCE, TransportDeleteByQueryAction.class),
                new ActionHandler<>(RethrottleAction.INSTANCE, TransportRethrottleAction.class));
    }

    @Override
    public List<NamedWriteableRegistry.Entry> getNamedWriteables() {
        return singletonList(
                new NamedWriteableRegistry.Entry(Task.Status.class, BulkByScrollTask.Status.NAME, BulkByScrollTask.Status::new));
    }

    @Override
    public List<RestHandler> getRestHandlers(Settings settings, RestController restController, ClusterSettings clusterSettings,
            IndexScopedSettings indexScopedSettings, SettingsFilter settingsFilter, IndexNameExpressionResolver indexNameExpressionResolver,
            Supplier<DiscoveryNodes> nodesInCluster) {
        return Arrays.asList(
                new RestReindexAction(settings, restController),
                new RestUpdateByQueryAction(settings, restController),
                new RestDeleteByQueryAction(settings, restController),
                new RestRethrottleAction(settings, restController, nodesInCluster));
    }

    @Override
    public Collection<Object> createComponents(Client client, ClusterService clusterService, ThreadPool threadPool,
                                               ResourceWatcherService resourceWatcherService, ScriptService scriptService,
                                               NamedXContentRegistry xContentRegistry, Environment environment,
                                               NodeEnvironment nodeEnvironment, NamedWriteableRegistry namedWriteableRegistry) {
        return Collections.singletonList(new ReindexSslConfig(environment.settings(), environment, resourceWatcherService));
    }

    @Override
    public List<Setting<?>> getSettings() {
        final List<Setting<?>> settings = new ArrayList<>();
        settings.add(TransportReindexAction.REMOTE_CLUSTER_WHITELIST);
        settings.addAll(ReindexSslConfig.getSettings());
        return settings;
    }
}
```