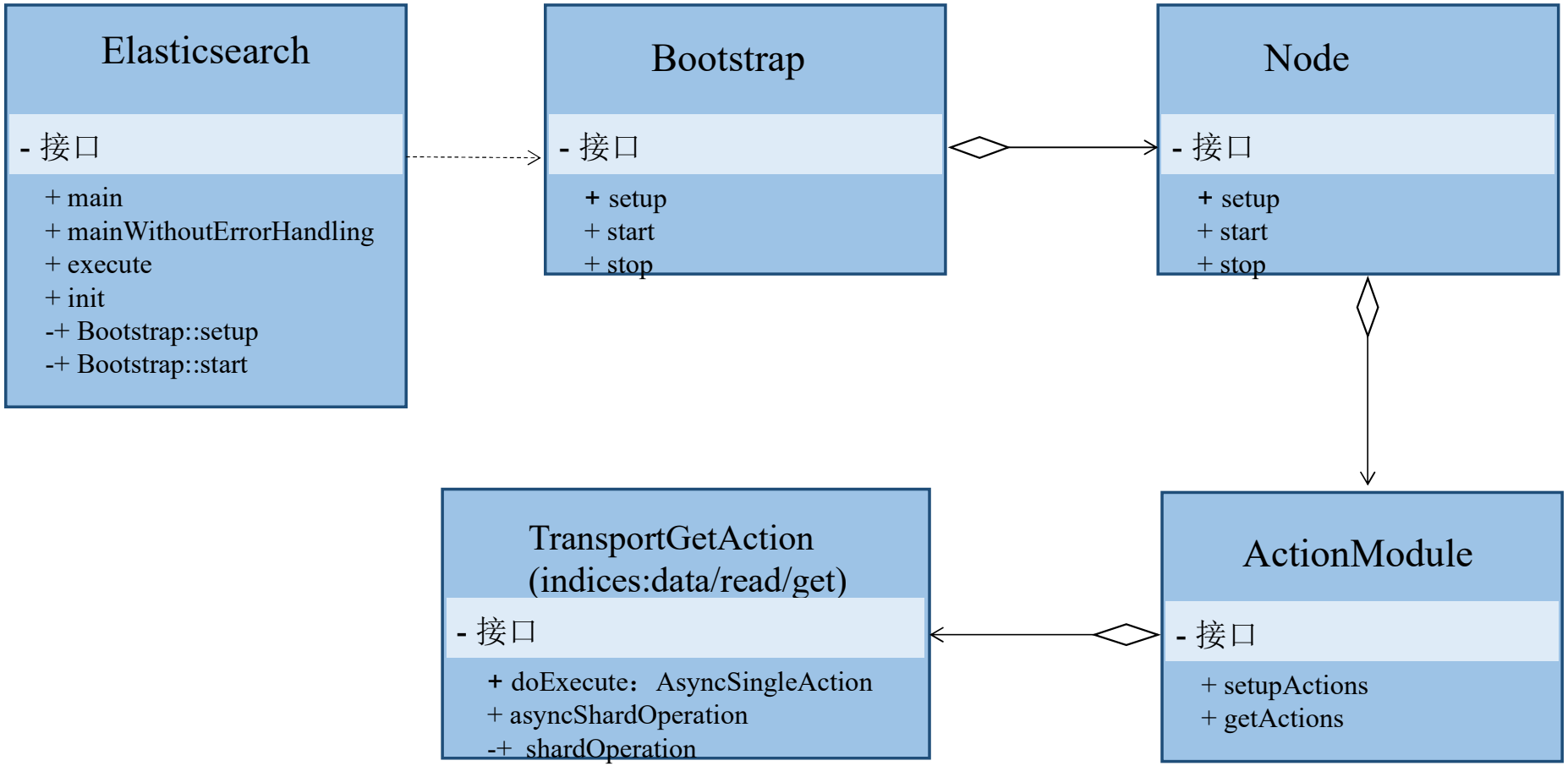
目 录

ES节点启动流程


Elasticsearch
- 接口
+ main
+ mainWithoutErrorHandling
+ execute
+ init
-+ Bootstrap::setup
-+ Bootstrap::start
Bootstrap
- 接口
+ setup
+ start
+ stop
Node
- 接口
+ setup
+ start
+ stop
ActionModule
- 接口
+ setupActions
+ getActions
TransportGetAction
(indices:data/read/get)
- 接口
+ doExecute：AsyncSingleAction
+ asyncShardOperation
-+ shardOperation

ES节点启动流程

Elasticsearch

- 接口

+ main
+ mainWithoutErrorHandling
+ execute
+ init
-+ Bootstrap::setup
-+ Bootstrap::start

Bootstrap

- 接口

+ init
+ start
-+ Node::start

处理命令行参数, 生成Environment

Bootstrap::init

Bootstrap::setup

启动modules目录下插件的native controller

initializeNatives & initializeProbes & addShutdownHook

Initializes SecurityManager

实例化ES Node，并设置启动前的环境和配置自检项

Bootstrap::start

Node::start

keepAliveThread.start()

```
final class Spawner void spawnNativeControllers(final Environment environment)
List<Path> paths = PluginsService.findPluginDirs(environment.modulesFile());
for (final Path modules : paths) {
    final PluginInfo info = PluginInfo.readFromProperties(modules);
    final Path spawnPath = Platforms.nativeControllerPath(modules);
    if (!Files.isRegularFile(spawnPath)) {...}
    if (!info.hasNativeController()) {...}
    final Process process = spawnNativeController(spawnPath, environment.tmpFile());
    processes.add(process);
}
```

```
final class Spawner private Process spawnNativeController(
command = spawnPath.toString();
final ProcessBuilder pb = new ProcessBuilder(command);
pb.start();
```

```
final class Bootstrap
private void start() throws N
    node.start();
    keepAliveThread.start();
}
```

```
keepAliveThread = new Thread( keepAliveLatch.await();
keepAliveThread.setDaemon(false);
Runtime.getRuntime().addShutdownHook(run() -> { keepAliveLatch.countDown(); });
```

ES节点启动流程

Node

- 接口

+ start

Node::start

更新节点状态

依次启动所有插件

依次启动所有服务:
IndicesService,
IndicesClusterStateService,
SnapshotsService,
SnapshotShardsService,
RoutingService,
SearchService,
MonitorService,
NodeConnectionsService,
ResourceWatcherService,
GatewayService,
TransportService,
Discovery,
ClusterService,
HttpServerTransport

依次触发所有插件的onNodeStarted

```
public class Node
```

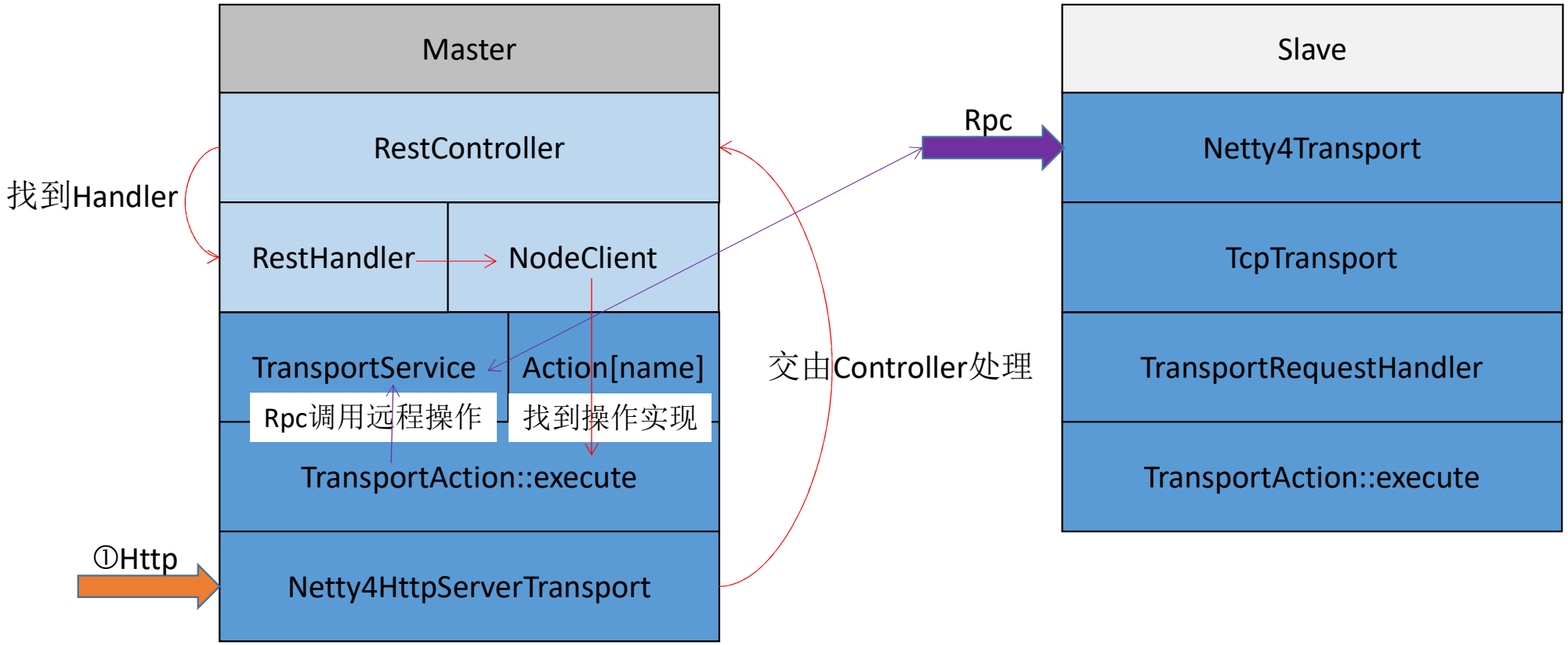
```
public Node start()
```

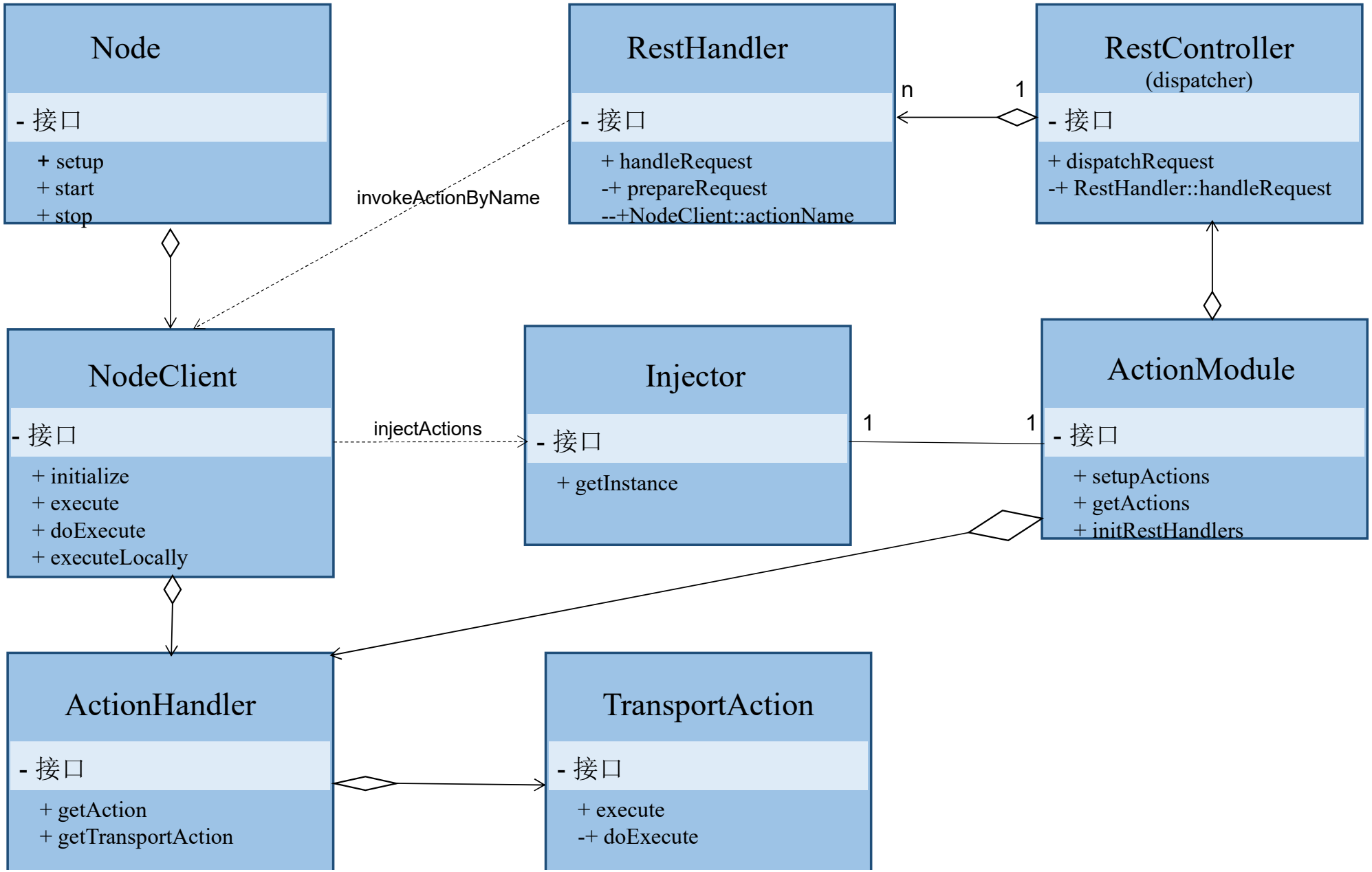
```
lifecycle.moveToStarted()
pluginLifecycleComponents.forEach(LifecycleComponent::start);
```

```
pluginLifecycleComponents.forEach(LifecycleComponent::start);

injector.getInstance(MappingUpdatedAction.class).setClient(client);
injector.getInstance(IndicesService.class).start();
injector.getInstance(IndicesClusterStateService.class).start();
injector.getInstance(SnapshotsService.class).start();
injector.getInstance(SnapshotShardsService.class).start();
injector.getInstance(RoutingService.class).start();
injector.getInstance(SearchService.class).start();
nodeService.getMonitorService().start();
```

处理Http和RPC请求



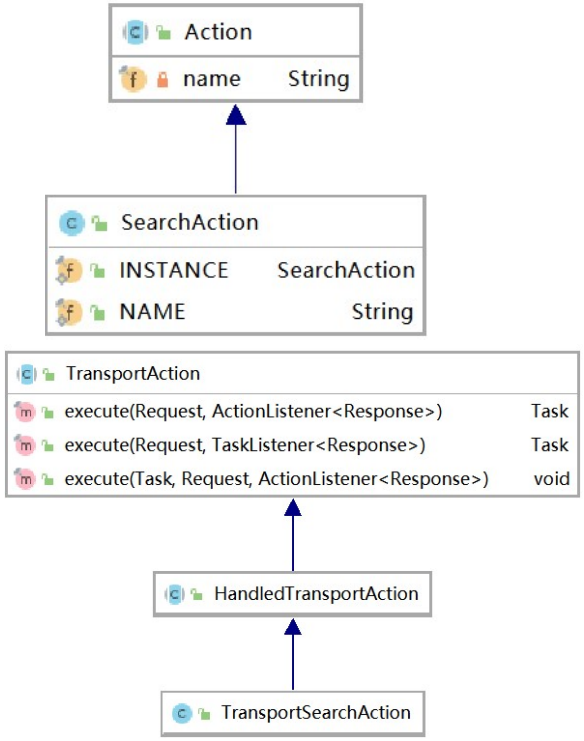

Node
- 接口
+ setup
+ start
+ stop
RestHandler
- 接口
+ handleRequest
-+ prepareRequest
--+NodeClient::actionName
n
1
RestController
(dispatcher)
- 接口
+ dispatchRequest
-+ RestHandler::handleRequest
invokeActionByName
NodeClient
- 接口
+ initialize
+ execute
+ doExecute
+ executeLocally
injectActions
Injector
- 接口
+ getInstance
1
1
ActionModule
- 接口
+ setupActions
+ getActions
+ initRestHandlers
ActionHandler
- 接口
+ getAction
+ getTransportAction
TransportAction
- 接口
+ execute
-+ doExecute

处理Http请求

ActionModule

- 接口

+ setupActions
+ getActions
+ initRestHandlers



Node(Environment, classpathPlugins)
--- 实例化ES节点的各类依赖

实例化ActionModule

注册各类TransportAction，用于处理ES操作

```
public class ActionModule
    Map<String, ActionHandler<?, ?>> setupActions(List<ActionPlugin> actionPlugins)
        actions.register(GetAction.INSTANCE, TransportGetAction.class);
        actions.register(BulkAction.INSTANCE, TransportBulkAction.class,
                TransportShardBulkAction.class);
        actions.register(SearchAction.INSTANCE, TransportSearchAction.class);
```

其中，每个Action对应了一个TransportAction，TransportAction实现了ES的各类具体操作。NodeClient可通过Action找到对应的TransportAction

同时注册各类插件内定义的TransportAction

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

HandledTransportAction可帮助子类实现向TransportService注册TransportAction的功能，可直接用TransportService通过Action::name调用对应的TransportAction

```
public class SearchAction
    public static final String NAME = "indices:data/read/search";
    private SearchAction() { super(NAME); }
```

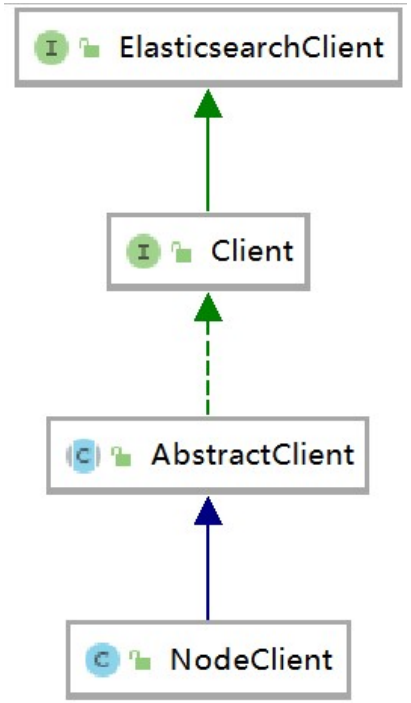
```
public abstract class HandledTransportAction
    transportService.registerRequestHandler(actionName, executor,
            new TransportHandler());
    class TransportHandler
        public final void messageReceived
            execute(task, request, new ChannelActionListener<>(channel, actionName, request));
```

处理Http请求：NodeClient（调用本地的TransportAction）

NodeClient

- 接口

+ initialize
+ execute
+ doExecute
+ executeLocally



Node(Environment, classpathPlugins)
--- 实例化ES节点的各类依赖

实例化ActionModule

实例化NodeClient

```
modules.add(b -> {  b.bind(NodeClient.class).toInstance(client);
```

为NodeClient注入Action->TransportAction依赖

```
client.initialize(injector.getInstance(new Key<Map<Action, TransportAction>>() {}),
public class NodeClient public void initialize(Map<Action, TransportAction> actions,
    this.actions = actions;
```

可通过传入Action调用对应的TransportAction

```
public abstract class AbstractClient
    public ActionFuture<GetResponse> get(final GetRequest request) {
        return execute(GetAction.INSTANCE, request);
    }
    void execute
        doExecute(action, request, listener);
public class NodeClient
    void doExecute
        executeLocally(action, request, listener);
    Task executeLocally
        return transportAction(action).execute(request, listener);
    TransportAction<Request, Response> transportAction(Action<Response> action)
        TransportAction<Request, Response> transportAction = actions.get(action);
```

处理Http请求

ActionModule

- 接口

+ setupActions
+ getActions
+ initRestHandlers

Node(Environment, classpathPlugins)
--- 实例化ES节点的各类依赖

实例化ActionModule

向RestController注册各类RestHandler，用于路由Http请求到TransportAction

```
public class ActionModule   public void initRestHandlers
    registerHandler.accept(new RestGetAction(settings, restController));
    registerHandler.accept(new RestSearchAction(settings, restController));
```

```
public class RestGetAction   public RestGetAction(final Settings settings, final RestController controller)
    controller.registerHandler(GET, path: "/{index}/_doc/{id}", handler: this);
    controller.registerHandler(HEAD, path: "/{index}/_doc/{id}", handler: this);
```

RestController调用RestHandler的handleRequest，在handleRequest中调用NodeClient的对应方法

```
public abstract class BaseRestHandler   public final void handleRequest
    final RestChannelConsumer action = prepareRequest(request, client);
        public class RestGetAction   public RestChannelConsumer prepareRequest
            return channel -> client.get(getRequest, new RestToXContentListener<GetResponse>(channel) {...});
    // execute the action
    action.accept(channel);
```

RestHandler	
handleRequest(RestRequest, RestChannel, NodeClient)	void
canTripCircuitBreaker()	boolean
supportsContentStream()	

BaseRestHandler	
getUsageCount()	long
getName()	String
handleRequest(RestRequest, RestChannel, NodeClient)	void

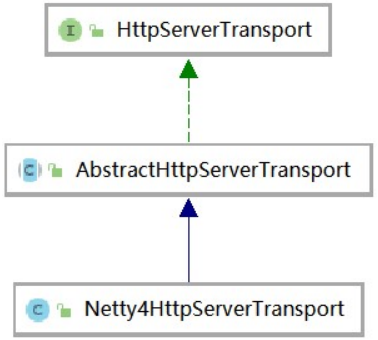
RestGetAction	
getName()	String
prepareRequest(RestRequest, NodeClient)	RestChannelConsumer

处理Http请求: 路由

RestController
implements Dispatcher

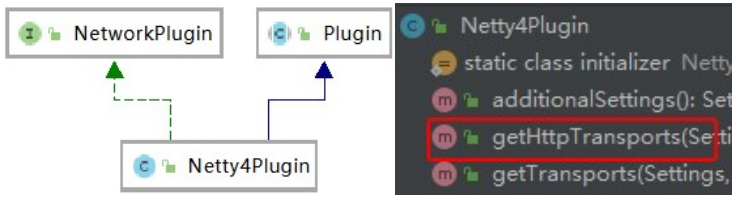
- 接口

+ registerHandler
+ dispatchRequest
-+ RestHandler::handleRequest




```
public abstract class AbstractHttpServerTransport
void dispatchRequest(final RestRequest restRequest, final RestChannel channel,
    dispatcher.dispatchRequest(restRequest, channel, threadContext);
```

```
public class RestController implements HttpServerTransport.Dispatcher
public void dispatchRequest(RestRequest request, RestChannel channel,
void tryAllHandlers(final RestRequest request, final RestChannel channel,
requestHandled = dispatchRequest(request, channel, client, mHandler);
final RestHandler wrappedHandler = mHandler.map(h -> handlerWra
wrappedHandler.handleRequest(request, responseChannel, client);
```




```
return Collections.singletonMap(NETTY_HTTP_T
    () -> new Netty4HttpServerTransport(sett
```

Node(Environment, classpathPlugins)
--- 实例化ES节点的各类依赖

实例化ActionModule

实例化RestController

注册各类RestHandler

将RestController注入NetworkModule

NetworkModule::getHttpServerTransportSupplier().get()
获取HttpServerTransport; 通过Netty4Plugin获取

HttpServerTransport::start

Http服务启动后，ES可将请求通过RestController，路由至对应的RestHandler，再经由RestHandler调用NodeClient对应操作，NodeClient根据操作的actionName找到对应的TransportAction，执行execute，得到执行结果

处理Http请求:Netty

处理Http请求:Netty配置与启动

Netty4HttpServerTransport	
settings()	Settings
doStart()	void
buildCorsConfig(Settings)	Netty4CorsConfig
bind(InetSocketAddress)	HttpServerChannel
stopInternal()	void
onException(HttpChannel, Exception)	void
configureServerChannelHandler()	ChannelHandler

ChannelInitializer	
initChannel(C)	void
channelRegistered(ChannelHandlerContext	void
exceptionCaught(ChannelHandlerContext, Throwable	void
handlerAdded(ChannelHandlerContext	void

HttpChannelHandler	
initChannel(Channel	void
exceptionCaught(ChannelHandlerContext, Throwable	void

doStart
--- 配置Netty::ServerBootstrap
--- 配置ChannelHandler
--- 启动服务

```
public class Netty4HttpServerTransport extends AbstractHttpServerTransport
public void start()
    protected void doStart()
        serverBootstrap = new ServerBootstrap();

        serverBootstrap.group(new NioEventLoopGroup(workerCount, daemonThreadFactory(settings,
            HTTP_SERVER_WORKER_THREAD_NAME_PREFIX)));
        serverBootstrap.channel(NioServerSocketChannel.class);
```

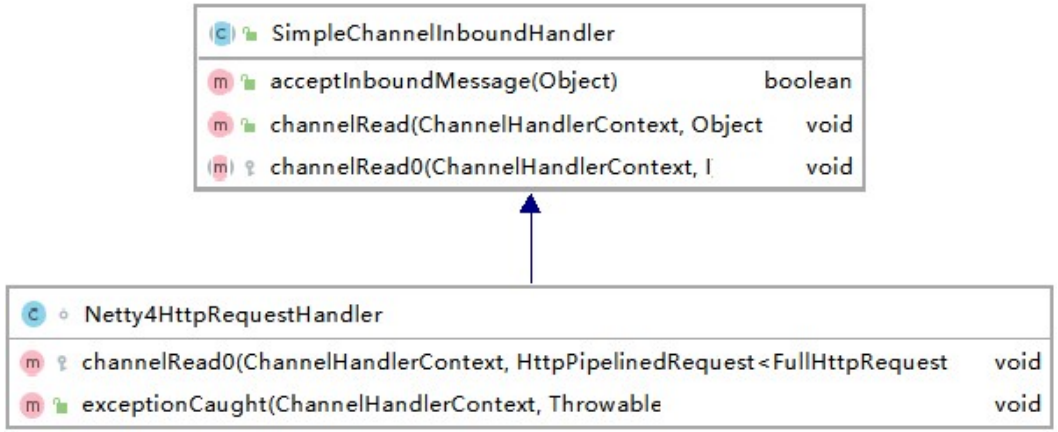
配置ChannelHandler

```
protected void doStart()
    serverBootstrap.childHandler(configureServerChannelHandler());
        public ChannelHandler configureServerChannelHandler() {
            return new HttpChannelHandler( transport: this, handlingSettings);
        }
protected static class HttpChannelHandler extends ChannelInitializer<Channel>
        protected void initChannel(Channel ch)
            ch.pipeline().addLast( name: "handler", requestHandler);
```

启动Http服务

```
public class Netty4HttpServerTransport extends AbstractHttpServerTransport protected HttpServerChannel bind(
    ChannelFuture future = serverBootstrap.bind(socketAddress).sync();
```

处理Http请求:InboundHandler

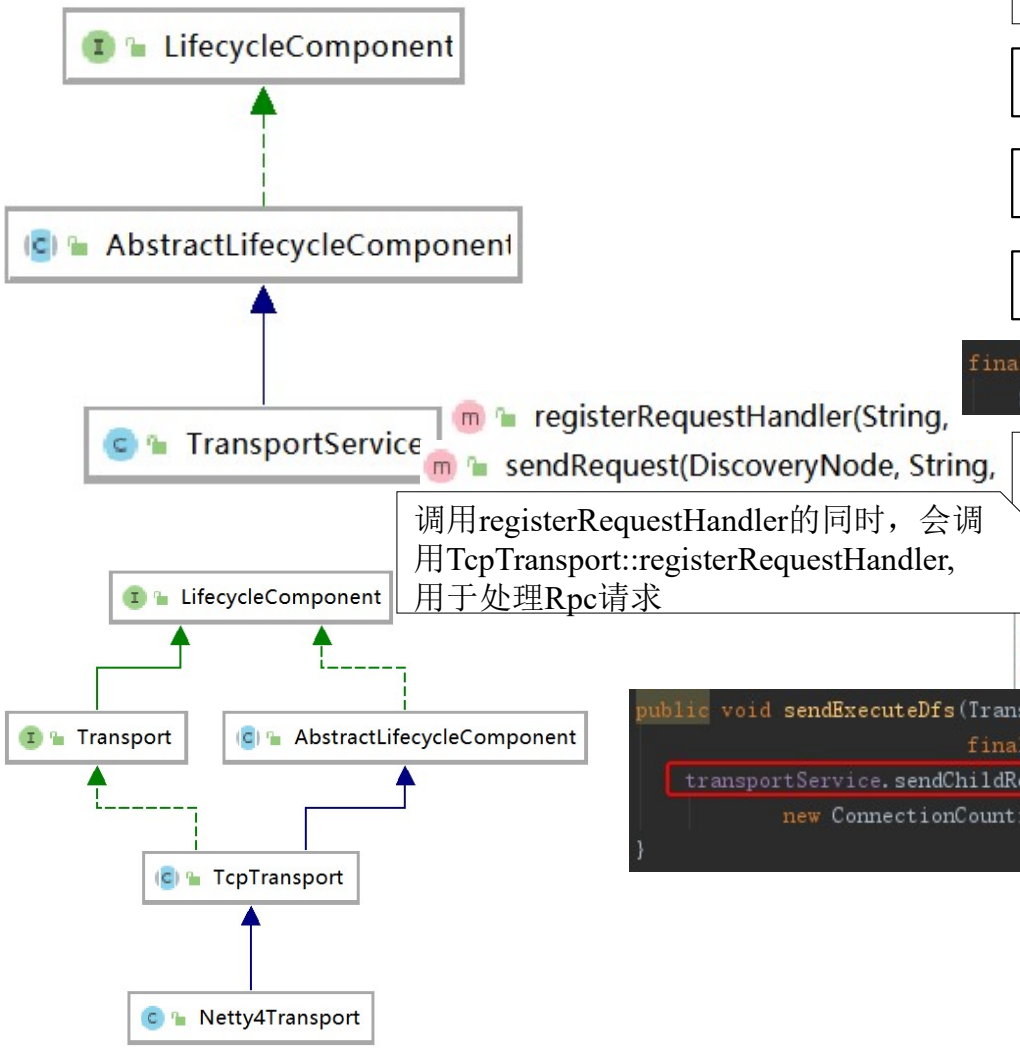



```
class Netty4HttpRequestHandler extends SimpleChannelInboundHandler<HttpPipelinedRequest<FullHttpRequest>>
    private final Netty4HttpServerTransport serverTransport;
    protected void channelRead0(ChannelHandlerContext ctx, HttpPipelinedRequest<FullHttpRequest> msg)
        serverTransport.incomingRequest(httpRequest, channel);

public abstract class AbstractHttpServerTransport extends AbstractLifecycleComponent implements HttpServerTransport
    public void incomingRequest(final HttpRequest httpRequest, final HttpChannel httpChannel)
        private void handleIncomingRequest(final HttpRequest httpRequest, final HttpChannel httpChannel, final Exception exception)
            void dispatchRequest(final RestRequest restRequest, final RestChannel channel, final Throwable badRequestCause)
                dispatcher.dispatchRequest(restRequest, channel, threadContext);
```

调用RestController::dispatchRequest, 执行请求

处理RPC请求:TransportService



调用registerRequestHandler的同时，会调用TcpTransport::registerRequestHandler，用于处理Rpc请求

Node(Environment, classpathPlugins)
--- 实例化ES节点的各类依赖

实例化TransportService

TransportService::start

实例化SearchTransportService

```
final SearchTransportService searchTransportService =  new SearchTransportService(transportService,
    SearchExecutionStatsCollector.makeWrapper(responseCollectorService));
```

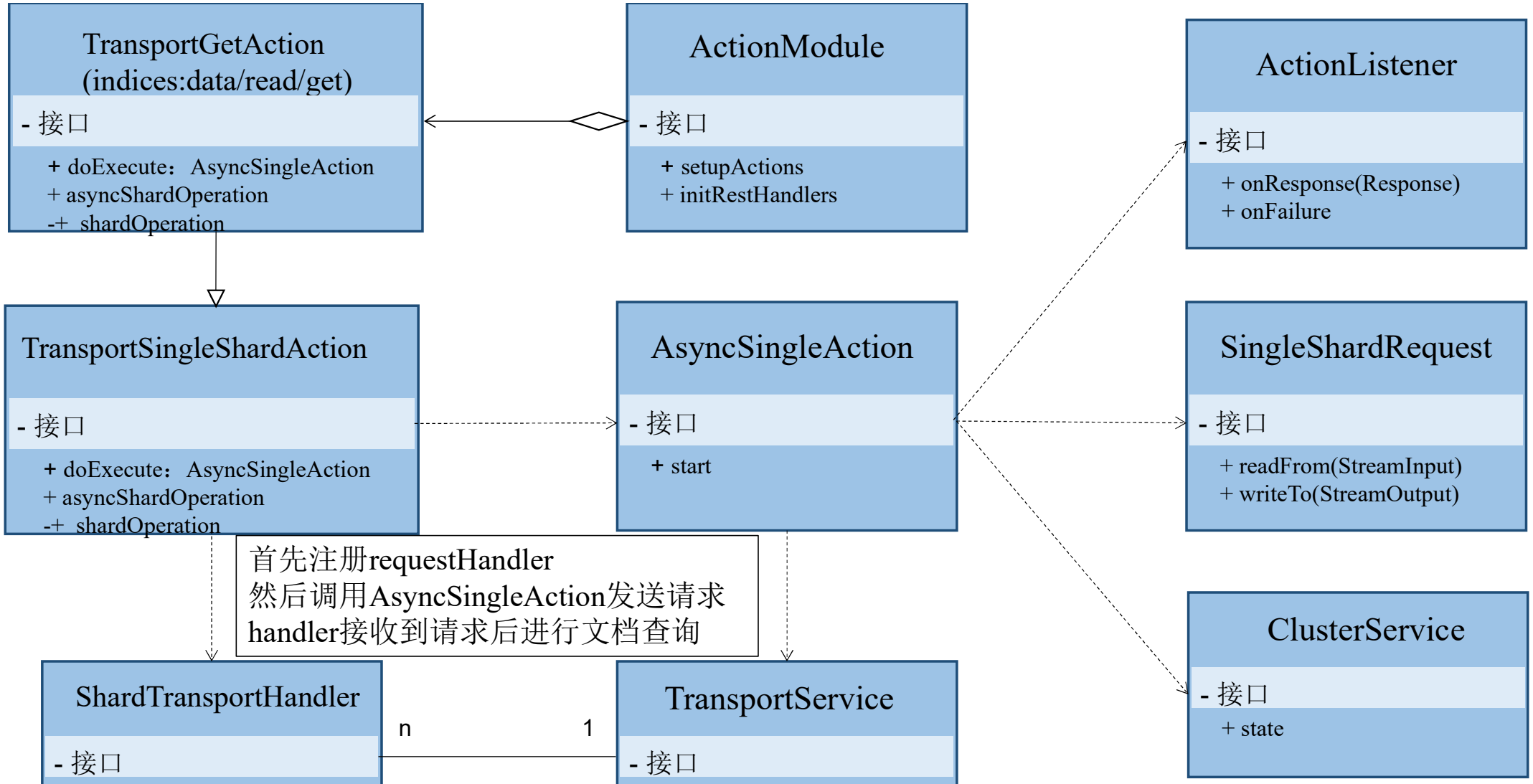
SearchTransportService封装了TransportService有关搜索的具体RPC调用

```
sendExecuteDfs(Connection, ShardSearchTransportRequest, SearchTask, SearchActionListener<DfsSearchResult>)
sendExecuteQuery(Connection, ShardSearchTransportRequest, SearchTask, SearchActionListener<SearchPhaseResult>)
sendExecuteQuery(Connection, QuerySearchRequest, SearchTask, SearchActionListener<QuerySearchResult>)
sendExecuteScrollQuery(Connection, InternalScrollSearchRequest, SearchTask, SearchActionListener<ScrollQuerySearchResult>)
```

```
public void sendExecuteDfs(Transport.Connection connection, final ShardSearchTransportRequest request, SearchTask task,
                           final SearchActionListener<DfsSearchResult> listener) {
    transportService.sendChildRequest(connection, DFS_ACTION_NAME, request, task,
            new ConnectionCountingHandler<>(listener, DfsSearchResult::new, clientConnections, connection.getNode().getId()));
}
```

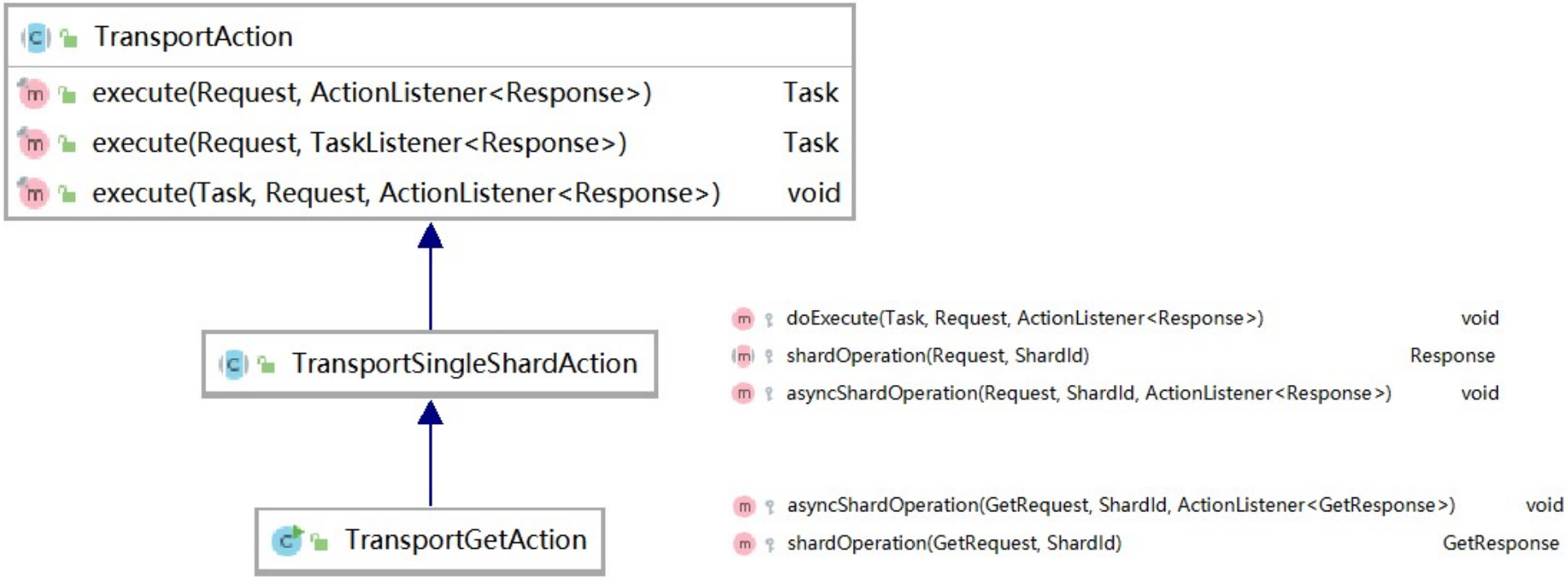
SearchTransportService::registerRequestHandler，注册每个RPC调用的具体实现

```
transportService.registerRequestHandler(DFS_ACTION_NAME, ThreadPool.Names.SAME,
```

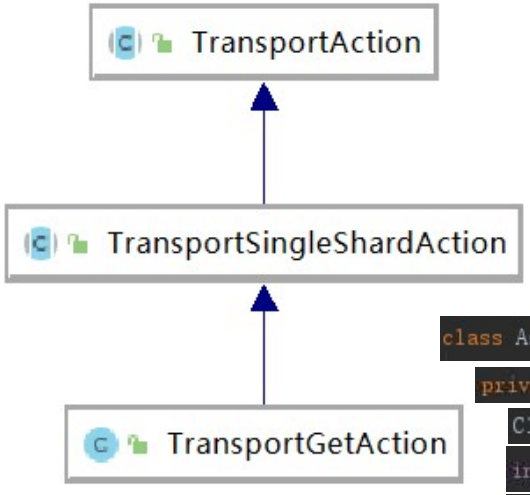

TransportGetAction
(indices:data/read/get)
- 接口
+ doExecute：AsyncSingleAction
+ asyncShardOperation
-+ shardOperation
ActionModule
- 接口
+ setupActions
+ initRestHandlers
ActionListener
- 接口
+ onResponse(Response)
+ onFailure
TransportSingleShardAction
- 接口
+ doExecute：AsyncSingleAction
+ asyncShardOperation
-+ shardOperation
AsyncSingleAction
- 接口
+ start
SingleShardRequest
- 接口
+ readFrom(StreamInput)
+ writeTo(StreamOutput)
首先注册requestHandler
然后调用AsyncSingleAction发送请求
handler接收到请求后进行文档查询
ClusterService
- 接口
+ state
ShardTransportHandler
- 接口
+ messageReceived
-+ asyncShardOperation
n
1
TransportService
- 接口
+ sendRequest
+ registerRequestHandler

处理文档Get请求

处理文档Get请求


TransportAction
execute(Request, ActionListener<Response>) Task
execute(Request, TaskListener<Response>) Task
execute(Task, Request, ActionListener<Response>) void
TransportSingleShardAction
doExecute(Task, Request, ActionListener<Response>) void
shardOperation(Request, ShardId) Response
asyncShardOperation(Request, ShardId, ActionListener<Response>) void
TransportGetAction
asyncShardOperation(GetRequest, ShardId, ActionListener<GetResponse>) void
shardOperation(GetRequest, ShardId) GetResponse

处理文档Get请求




```
class AsyncSingleAction
  private AsyncSingleAction(Request request,
    ClusterBlockException blockException = checkGlobalBlock(clusterState);
    indexNameExpressionResolver.concreteSingleIndex(clusterSta
    protected void resolveRequest
      state.metaData().resolveIndexRouting(r
    blockException = checkRequestBlock(clusterState, internalRequest);
    this.shardIt = shards(clusterState, internalRequest);
  public void start() {
    if (shardIt == null) {
      // just execute it on the local node
      transportService.sendRequest(clusterService.localNode()
    else {
      perform( currentFailure: null);
```

TransportAction::execute
--- 在TaskManager中注册此次任务
--- 应用ActionFilter
--- 将请求转至子类doExecute

TransportSingleShardAction::doExecute

实例化内部类AsyncSingleAction

检测集群全局是否读阻塞

解析获取索引实际名称(Alias)

解析获取此次操作的routing值

检测此索引是否设置了读阻塞

根据docId，routing，preference计算分片shardIt

AsyncSingleAction::start

若shardIt为null, 往本地节点发送docGet请求(本地调用)

文档Get操作对应的actionName为indices:data/read/get[s]
具体操作定义于内部类ShardTransportHandler中，
实际调用TransportSingleShardAction::asyncShardOperation

处理文档Get请求

TransportSingleShardAction	
isSubAction()	boolean
doExecute(Task, Request, ActionListener<Response>)	void
shardOperation(Request, ShardId)	Response
asyncShardOperation(Request, ShardId, ActionListener<Response>)	void
newResponse()	Response
resolveIndex(Request)	boolean
checkGlobalBlock(ClusterState)	ClusterBlockException
checkRequestBlock(ClusterState, InternalRequest)	ClusterBlockException
resolveRequest(ClusterState, InternalRequest)	void
shards(ClusterState, InternalRequest)	ShardsIterator
getExecutor(Request, ShardId)	String

AsyncSingleAction	
start()	void
onFailure(ShardRouting, Exception)	void
perform(Exception)	void

```
class AsyncSingleAction private void perform(
final ShardRouting shardRouting = shardIt.nextOrNull();
```

```
DiscoveryNode node = nodes.get(shardRouting.currentNodeId());
```

```
transportService.sendRequest(node, transportShardAction,
```

TransportAction::execute
--- 在TaskManager中注册此次任务
--- 应用ActionFilter
--- 将请求转至子类doExecute

TransportSingleShardAction::doExecute

AsyncSingleAction::start

若shardIt不为null

获取具体一个主/副分片shardRouting

获取分片所在的节点

向目标节点送文档Get Rpc请求

TransportGetAction::asyncShardOperation
--- 执行具体的文档查询操作

处理文档Get请求

TransportGetAction	
resolveIndex(GetRequest)	boolean
shards(ClusterState, InternalRequest)	ShardIterator
resolveRequest(ClusterState, InternalRequest)	void
asyncShardOperation(GetRequest, ShardId, ActionListener<GetResponse>)	void
shardOperation(GetRequest, ShardId)	GetResponse
newResponse()	GetResponse
getExecutor(GetRequest, ShardId)	String

```
public class TransportGetAction void asyncShardOperation(GetRequest request, ShardId shardId,
  IndexService indexService = indicesService.indexServiceSafe(shardId.getIndex());
  IndexShard indexShard = indexService.getShard(shardId.id());
  if (request.realtime()) {
    super.asyncShardOperation(request, shardId, listener);
    abstract class TransportSingleShardAction protected void asyncShardOperation
      listener.onResponse(shardOperation(request, shardId));
protected GetResponse shardOperation(GetRequest request, ShardId shardId)
  if (request.refresh() && !request.realtime()) {
      indexShard.refresh( source: "refresh_flag_get");
  }
  GetResult result = indexShard.getService().get(request.type(), request.id(), request.storedFields(),
          request.realtime(), request.version(), request.versionType(), request.fetchSourceContext());
  return new GetResponse(result);
```

TransportGetAction::asyncShardOperation
--- 执行具体的文档查询操作

根据indexName和shardId获取IndexService和IndexShard

非实时搜索，request.realtime == false

为indexShard添加refresh listener，分片刷新后触发，调用TransportGetAction::shardOperation

若listener槽位耗尽，直接触发refresh

TransportGetAction::shardOperation

设置了强制刷新，request.refresh==true, realtime == false

IndexShard::refresh, 刷新分片，重新获取IndexReader

ShardGetService::get，获取文档

AsyncSingleAction

- 作用

  + 选择一个可用于执行后续操作的分片，并发送请求

- 流程

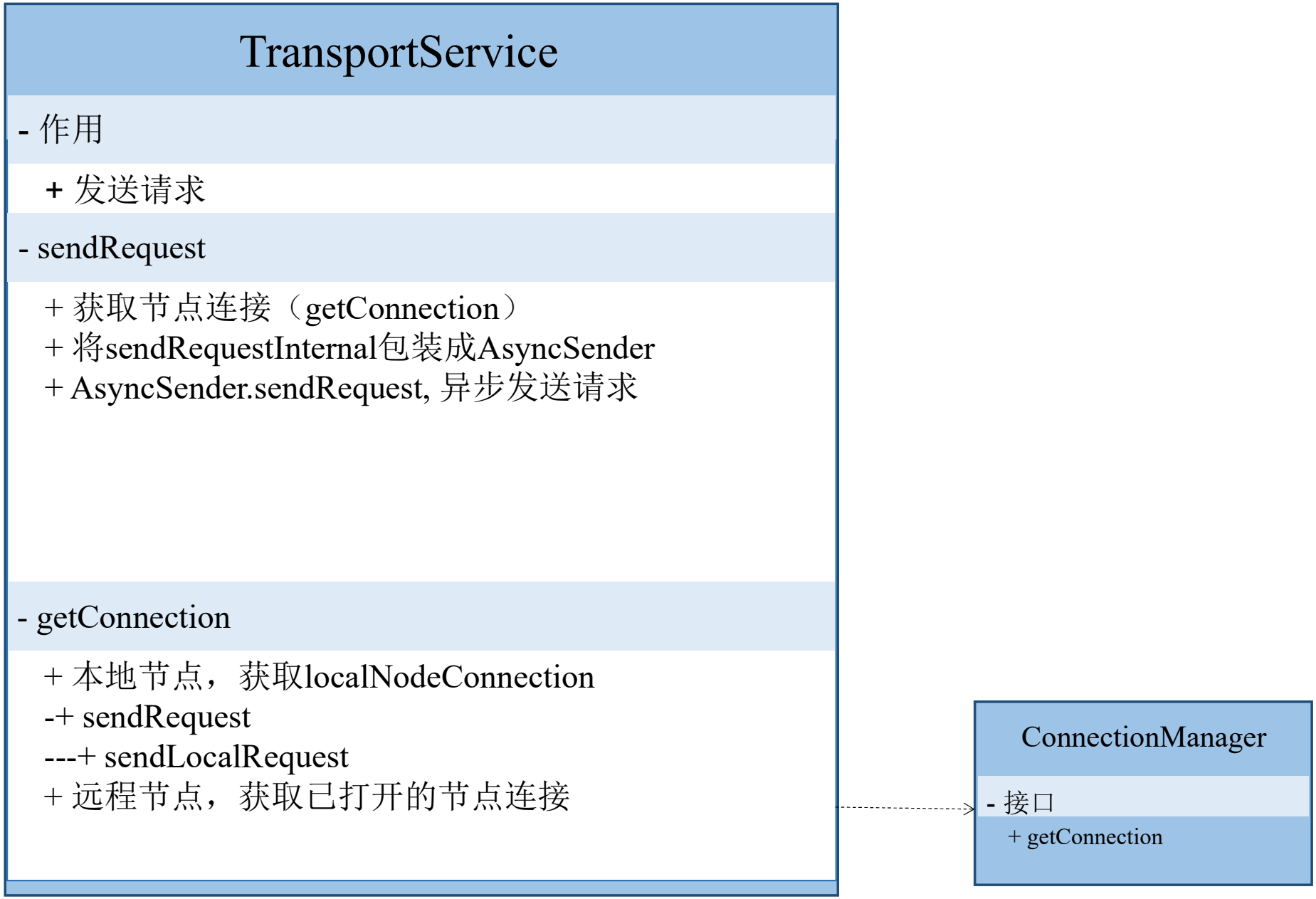
  + 获取集群状态（ClusterState）
  + 获取节点列表（DiscoveryNodes）
  + 解析索引名（循环解析Alias）
  + 解析Routing（Alias routing）
  + 检测索引是否Blocked（抛异常）
  + 计算ShardId（ShardIterator，本地null）
  + 发送读请求

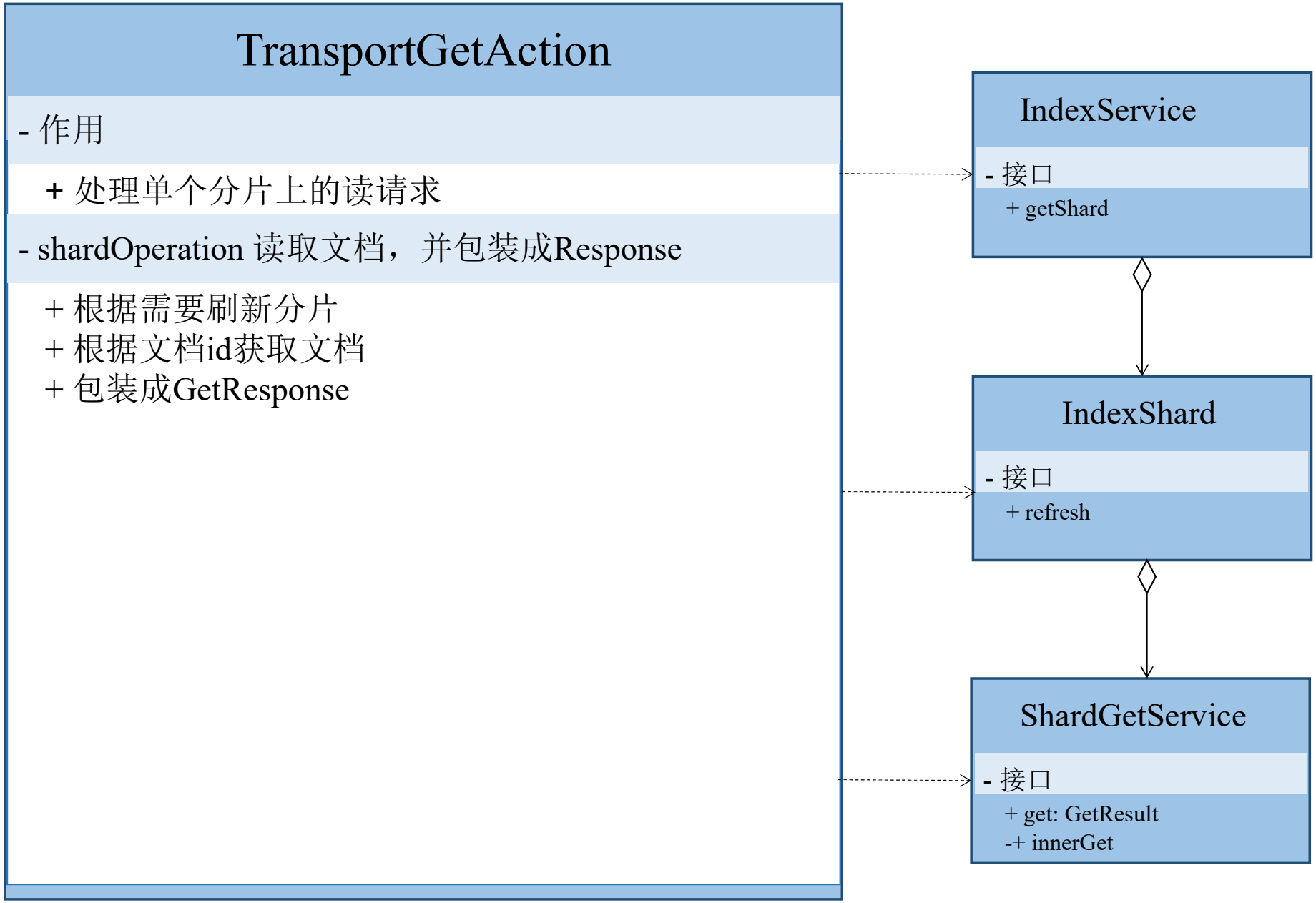
- 本地读请求

  + TransportService.send(localNode, action, request)

- 远程读请求

  + 选择一个分片，获取nodeId，获取node
  + TransportService.send(node, action, request)


TransportService
- 作用
+ 发送请求
- sendRequest
+ 获取节点连接（getConnection）
+ 将sendRequestInternal包装成AsyncSender
+ AsyncSender.sendRequest, 异步发送请求
- getConnection
+ 本地节点，获取localNodeConnection
-+ sendRequest
---+ sendLocalRequest
+ 远程节点，获取已打开的节点连接
ConnectionManager
- 接口
+ getConnection


TransportGetAction
- 作用
+ 处理单个分片上的读请求
- shardOperation 读取文档，并包装成Response
+ 根据需要刷新分片
+ 根据文档id获取文档
+ 包装成GetResponse
IndexService
- 接口
+ getShard
IndexShard
- 接口
+ refresh
ShardGetService
- 接口
+ get: GetResult
-+ innerGet

ShardGetService

- 作用

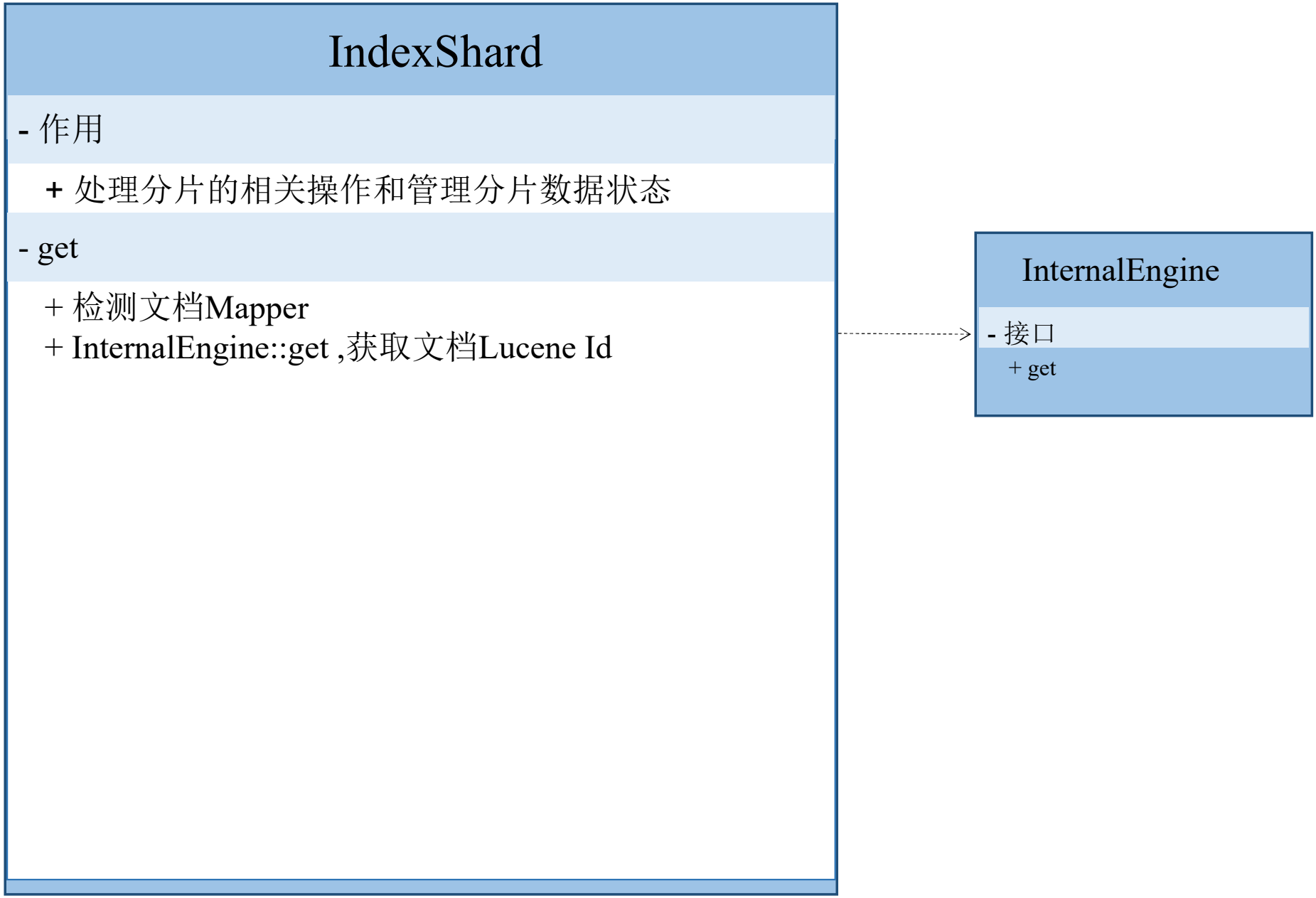
  + 根据文档id获取文档

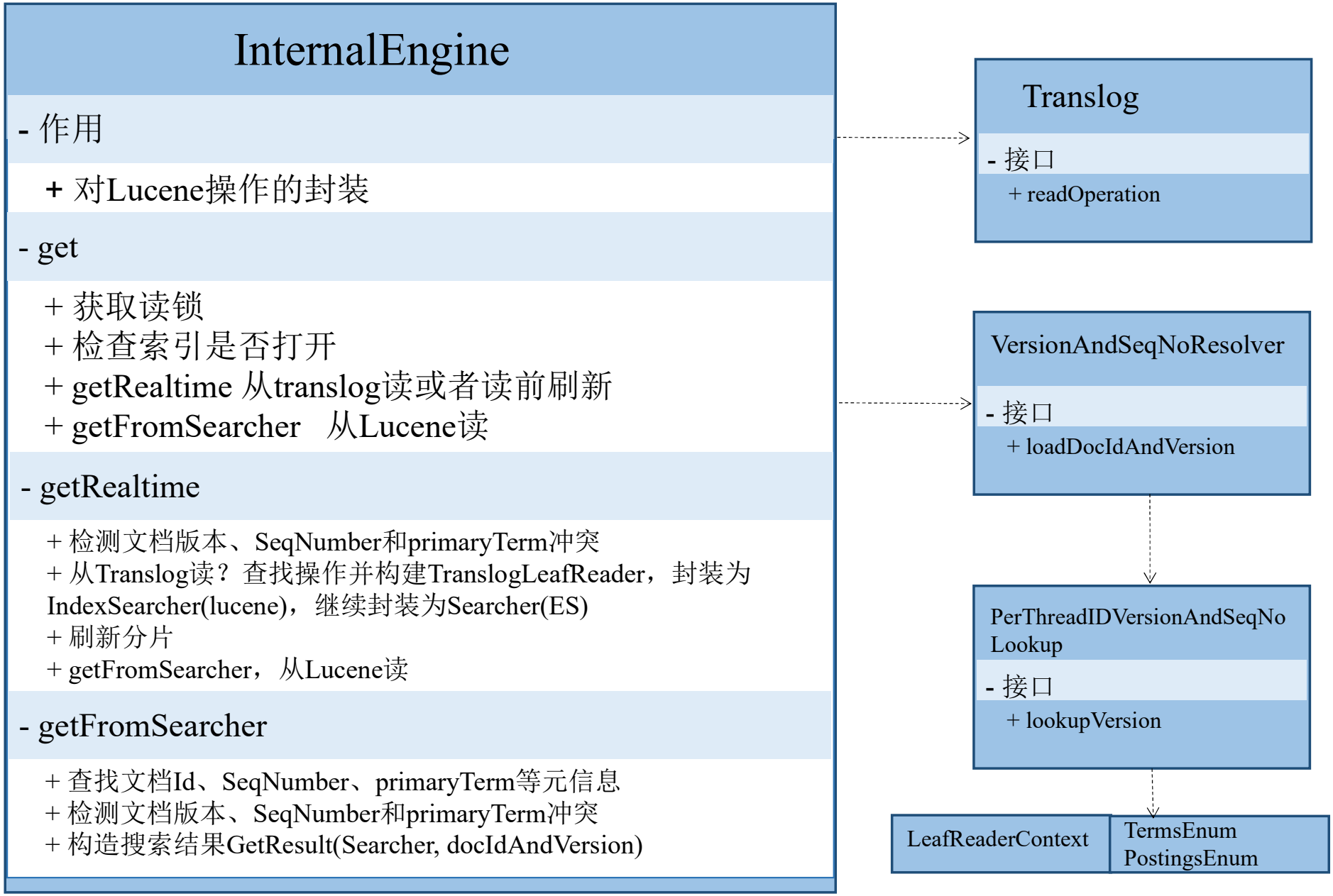
- innerGet

  + 判断是否包含_source
  + 确定type
  + IndexShard::get 获取Lucene内部文档Id
  + innerGetLoadFromStoredFields 获取文档内容

- innerGetLoadFromStoredFields 获取数据，字段过滤

  + 根据文档Id（Lucene）和字段，获取文档内容
  --+ LeafReader::document(docId, StoredFieldVisitor)
  + 获取source，封装成GetResult


IndexShard
- 作用
+ 处理分片的相关操作和管理分片数据状态
- get
+ 检测文档Mapper
+ InternalEngine::get ,获取文档Lucene Id
InternalEngine
- 接口
+ get


InternalEngine
- 作用
+ 对Lucene操作的封装
- get
+ 获取读锁
+ 检查索引是否打开
+ getRealtime 从translog读或者读前刷新
+ getFromSearcher 从Lucene读
- getRealtime
+ 检测文档版本、SeqNumber和primaryTerm冲突
+ 从Translog读？查找操作并构建TranslogLeafReader，封装为IndexSearcher(lucene)，继续封装为Searcher(ES)
+ 刷新分片
+ getFromSearcher，从Lucene读
- getFromSearcher
+ 查找文档Id、SeqNumber、primaryTerm等元信息
+ 检测文档版本、SeqNumber和primaryTerm冲突
+ 构造搜索结果GetResult(Searcher, docIdAndVersion)
Translog
- 接口
+ readOperation
VersionAndSeqNoResolver
- 接口
+ loadDocIdAndVersion
PerThreadIDVersionAndSeqNoLookup
- 接口
+ lookupVersion
LeafReaderContext
TermsEnum
PostingsEnum

Elasticsearch搜索类型：

1、DFS_QUERY_AND_FETCH

2、DFS_QUERY_THEN_FTECH

3、QUERY_AND_FETCH

4、QUERY_THEN_FETCH

Query-Then-Fetch:

1、发送查询到每个shard

2、找到所有匹配的文档，并使用本地的Term/Document Frequency信息进行打分

3、对结果构建一个优先队列（排序，标页等）

4、返回关于结果的元数据到请求节点（不包含文档数据）

5、来自所有shard的分数合并起来，并在请求节点上进行排序，文档被按照查询要求进行选择

6、实际文档从他们各自所在的独立的shard上检索出来

（每个分片的Term/Document frequency统计信息不一致，文档打分存在不应有的差异）

DFS-Query-Then-Fetch:

1、预查询每个shard，询问Term和Document frequency

2、发送查询到每隔shard

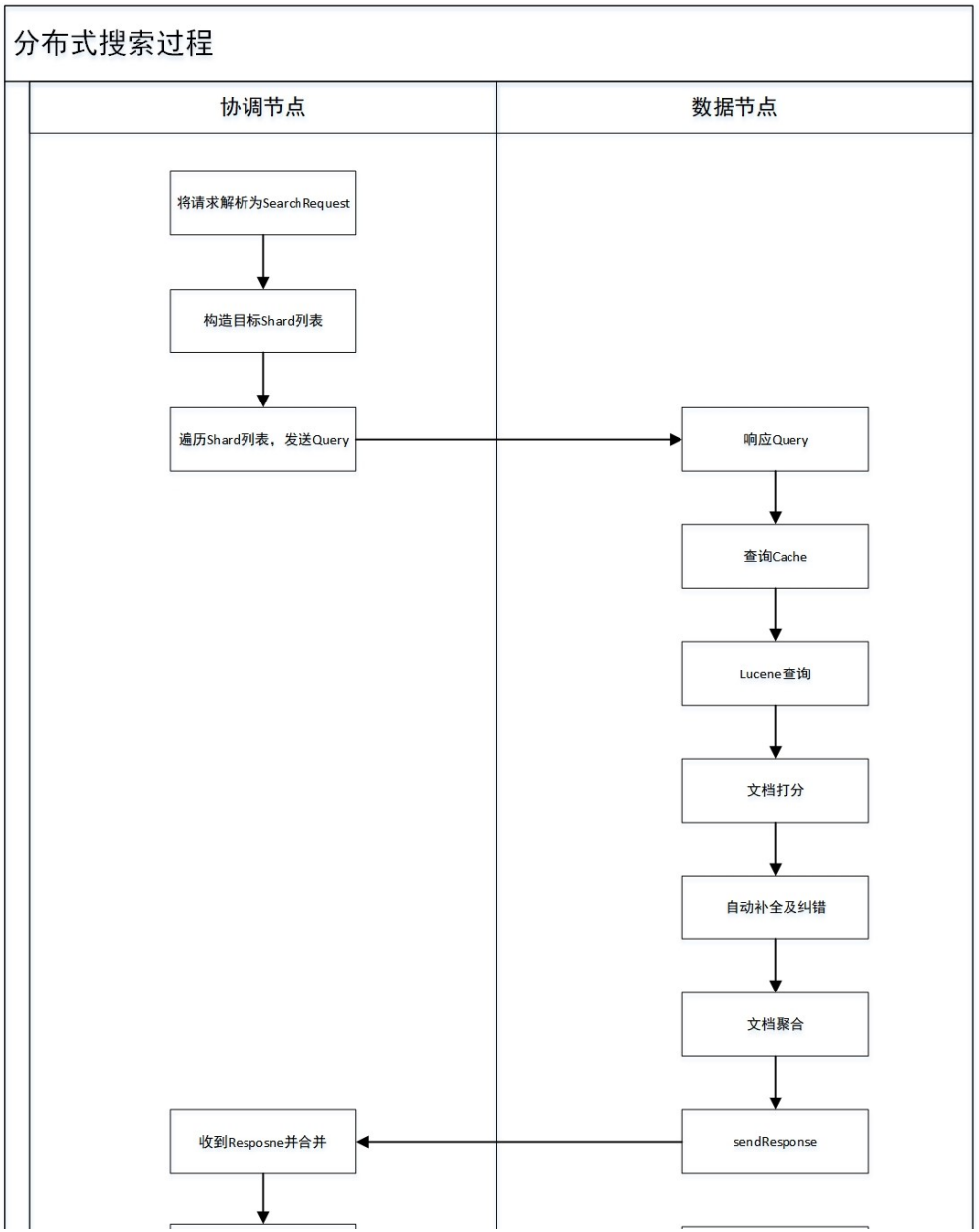
3、找到所有匹配的文档，并使用全局的Term/Document Frequency信息进行打分

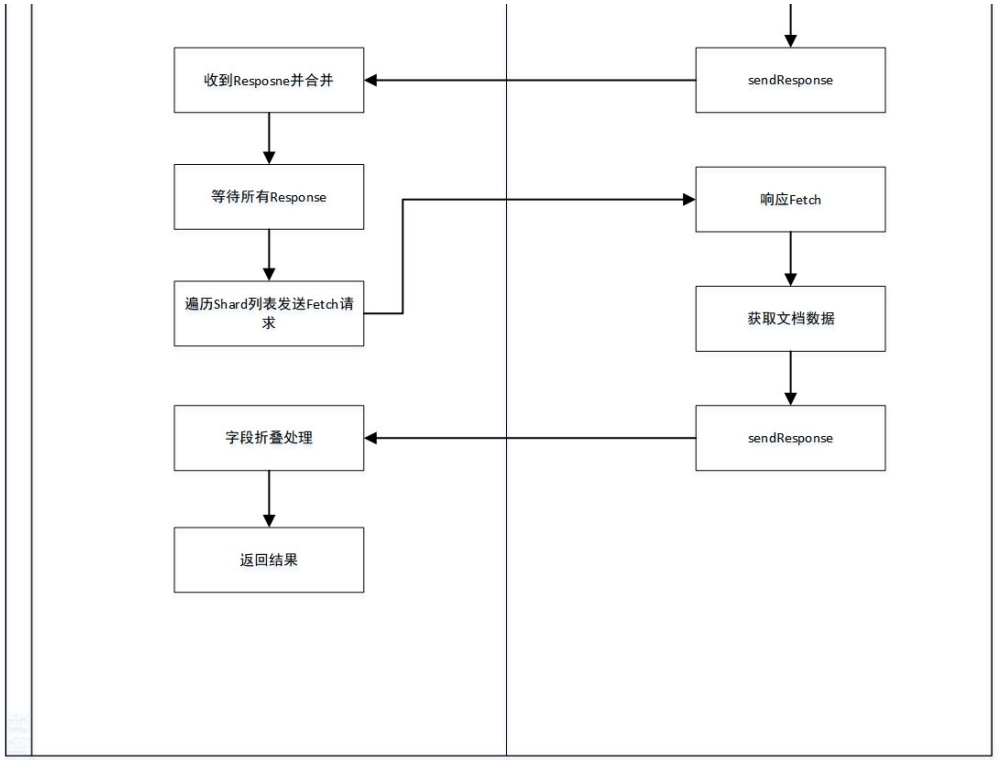
4、对结果构建一个优先队列（排序，标页等）

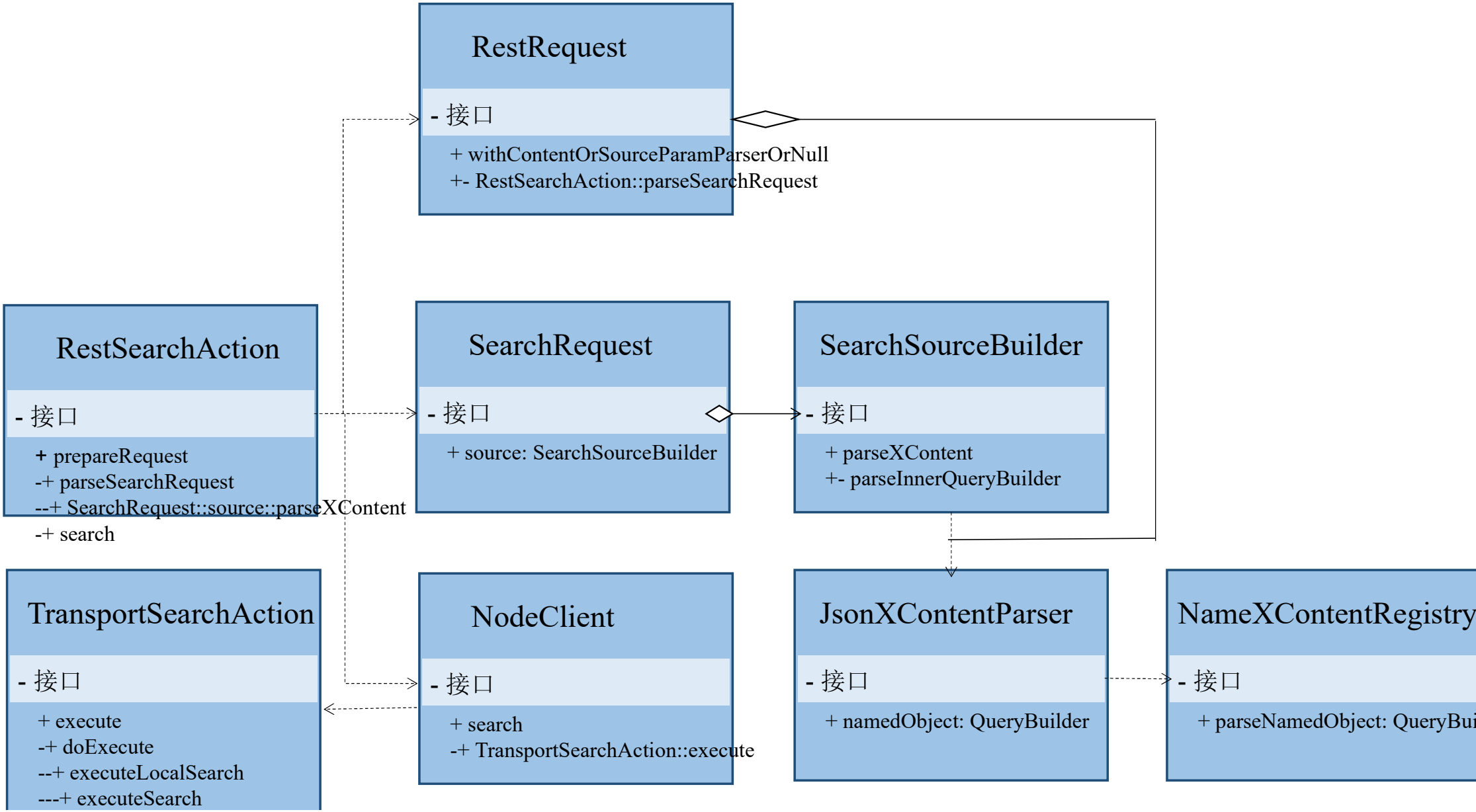
5、返回关于结果的元数据到请求节点。注意，实际文档还没有发送，只是分数

6、来自所有shard的分数合并起来，并在请求节点上进行排序，文档被按照查询要求进行选择

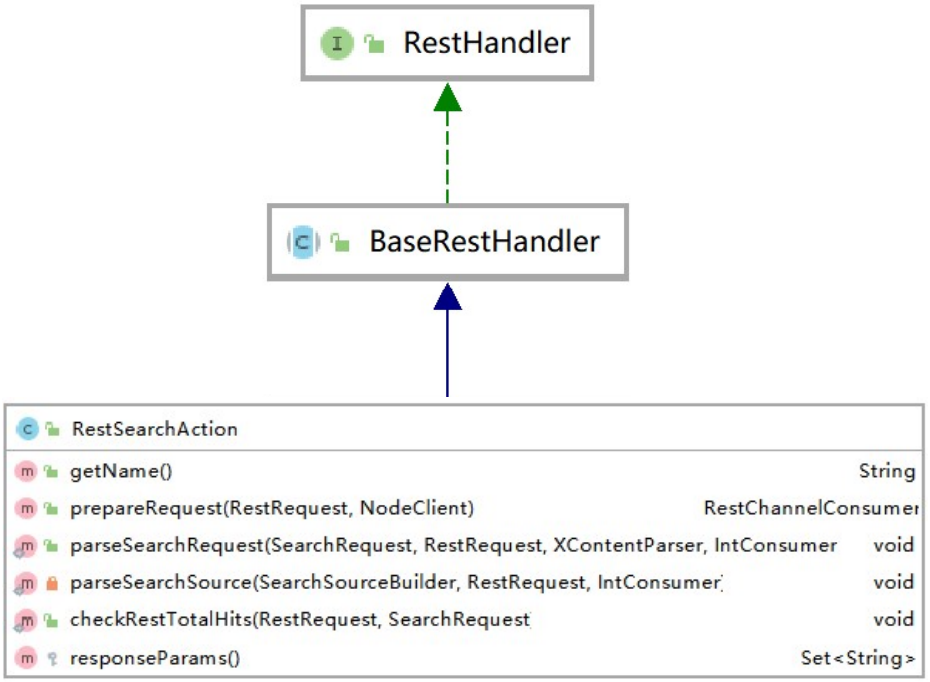
7、实际文档从他们各自所在的独立的shard上检索出来


分布式搜索过程
协调节点
数据节点
将请求解析为SearchRequest
构造目标Shard列表
遍历Shard列表，发送Query
响应Query
查询Cache
Lucene查询
文档打分
自动补全及纠错
文档聚合
sendResponse
收到Resposne并合并


收到Resposne并合并
等待所有Response
遍历Shard列表发送Fetch请求
字段折叠处理
返回结果
sendResponse
响应Fetch
获取文档数据
sendResponse


RestRequest
- 接口
+ withContentOrSourceParamParserOrNull
+- RestSearchAction::parseSearchRequest
RestSearchAction
- 接口
+ prepareRequest
-+ parseSearchRequest
--+ SearchRequest::source::parseXContent
-+ search
SearchRequest
- 接口
+ source: SearchSourceBuilder
SearchSourceBuilder
- 接口
+ parseXContent
+- parseInnerQueryBuilder
TransportSearchAction
- 接口
+ execute
-+ doExecute
--+ executeLocalSearch
---+ executeSearch
NodeClient
- 接口
+ search
-+ TransportSearchAction::execute
JsonXContentParser
- 接口
+ namedObject: QueryBuilder
NameXContentRegistry
- 接口
+ parseNamedObject: QueryBui

处理文档Search请求



RestSearchAction::execute
--- 先解析查询请求
--- 再执行TransportSearchAction::execute，开始搜索操作

BaseRestHandler::handleRequest

RestSearchAction::prepareRequest

RestSearchAction::parseSearchRequest

```
searchRequest.source().parseXContent(requestContentParser, checkTrailingTokens: true);
```

处理请求的各类参数，从RestRequest构造SearchRequest

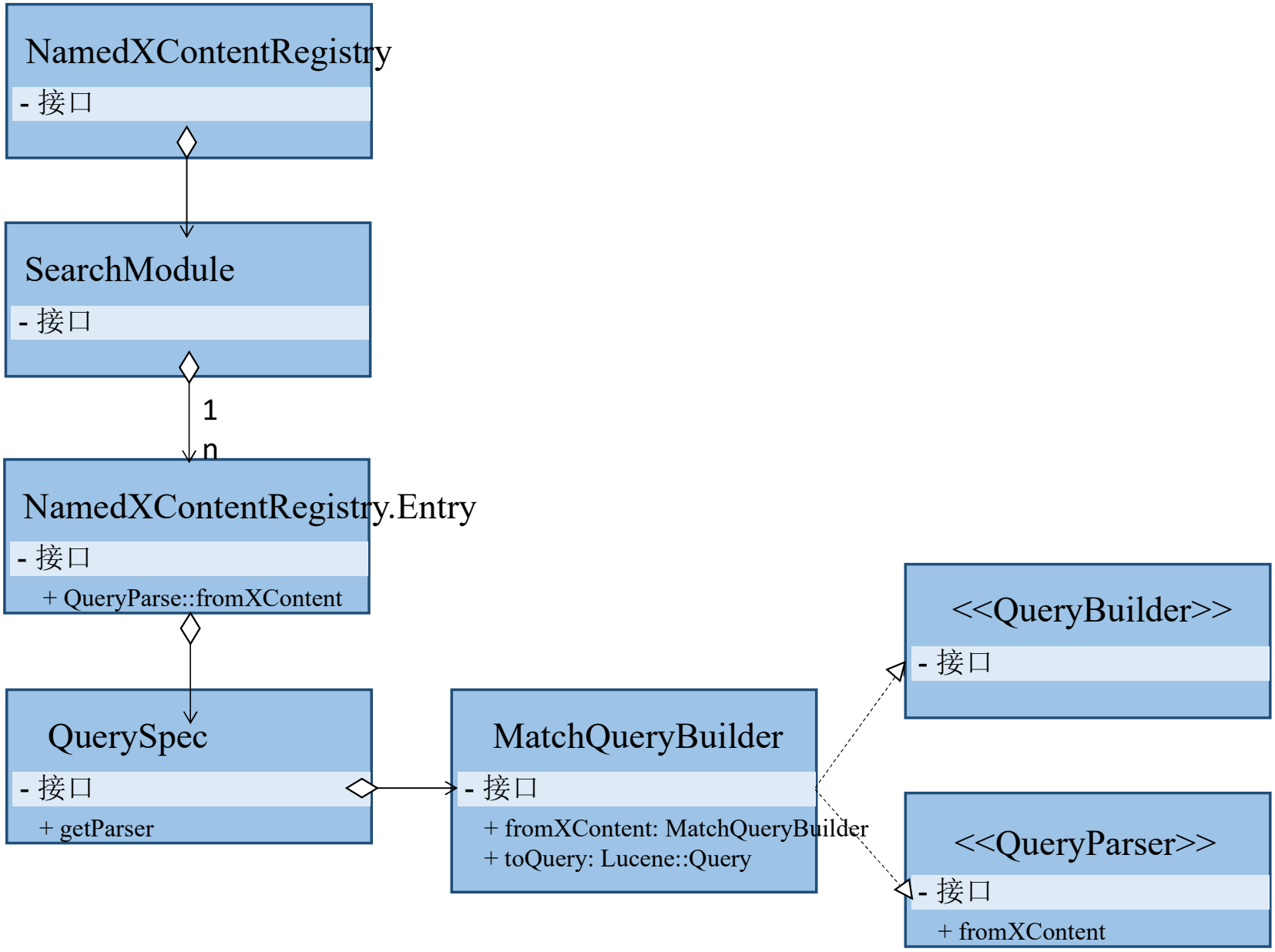
```
searchRequest.setMaxConcurrentShardRequests(maxConcurrentShardRequests);
searchRequest.allowPartialSearchResults(request.paramAsBoolean(
```

```
String searchType = request.param( key: "search_type");
if ("query_and_fetch".equals(searchType) ||
        "dfs_query_and_fetch".equals(searchType)) {
    throw new IllegalArgumentException("Unsupported search type [" + searchType + "]");
} else {
    searchRequest.searchType(searchType);
}
```

```
searchRequest.routing(request.param( key: "routing"));
searchRequest.preference(request.param( key: "preference"));
```

通过NodeClient调用TransportSearchAction::execute，开始搜索

```
return channel -> client.search(searchRequest, new RestStatusToXContentListener<>(channel));
```


NamedXContentRegistry
- 接口
SearchModule
- 接口
1
n
NamedXContentRegistry.Entry
- 接口
+ QueryParse::fromXContent
QuerySpec
- 接口
+ getParser
MatchQueryBuilder
- 接口
+ fromXContent: MatchQueryBuilder
+ toQuery: Lucene::Query
<<QueryBuilder>>
- 接口
<<QueryParser>>
- 接口
+ fromXContent

查询请求解析

NamedXContentRegistry.java

```
registry = {Collections$UnmodifiableMap@14454}  size = 16
  0 = {Collections$UnmodifiableMap$UnmodifiableEntrySet$UnmodifiableEntry@14673} "interface org.elasticsearch.persistent.PersistentTaskState" -> " size = 3"
  1 = {Collections$UnmodifiableMap$UnmodifiableEntrySet$UnmodifiableEntry@14674} "interface org.elasticsearch.tasks.Task$Status" -> " size = 1"
  2 = {Collections$UnmodifiableMap$UnmodifiableEntrySet$UnmodifiableEntry@14675} "interface org.elasticsearch.search.aggregations.BaseAggregationBuilder" -> " size = 56"
  3 = {Collections$UnmodifiableMap$UnmodifiableEntrySet$UnmodifiableEntry@14676} "interface org.elasticsearch.index.query.QueryBuilder" -> " size = 50"
  4 = {Collections$UnmodifiableMap$UnmodifiableEntrySet$UnmodifiableEntry@14677} "class org.elasticsearch.search.rescore.RescorerBuilder" -> " size = 1"
  5 = {Collections$UnmodifiableMap$UnmodifiableEntrySet$UnmodifiableEntry@14678} "class org.elasticsearch.index.query.functionscore.ScoreFunctionBuilder" -> " size = 6"
  6 = {Collections$UnmodifiableMap$UnmodifiableEntrySet$UnmodifiableEntry@14679} "interface org.elasticsearch.persistent.PersistentTaskParams" -> " size = 4"
  7 = {Collections$UnmodifiableMap$UnmodifiableEntrySet$UnmodifiableEntry@14680} "class org.elasticsearch.xpack.core.ccr.ShardFollowNodeTaskStatus" -> " size = 1"
  8 = {Collections$UnmodifiableMap$UnmodifiableEntrySet$UnmodifiableEntry@14681} "interface org.elasticsearch.index.rankeval.MetricDetail" -> " size = 4"
  9 = {Collections$UnmodifiableMap$UnmodifiableEntrySet$UnmodifiableEntry@14682} "interface org.elasticsearch.cluster.routing.allocation.command.AllocationCommand" -> " size = 5"
  10 = {Collections$UnmodifiableMap$UnmodifiableEntrySet$UnmodifiableEntry@14683} "interface org.elasticsearch.xpack.core.indexlifecycle.LifecycleType" -> " size = 1"
  11 = {Collections$UnmodifiableMap$UnmodifiableEntrySet$UnmodifiableEntry@14684} "class org.elasticsearch.search.suggest.SuggestionBuilder" -> " size = 3"
  12 = {Collections$UnmodifiableMap$UnmodifiableEntrySet$UnmodifiableEntry@14685} "class org.elasticsearch.action.admin.indices.rollover.Condition" -> " size = 3"
  13 = {Collections$UnmodifiableMap$UnmodifiableEntrySet$UnmodifiableEntry@14686} "interface org.elasticsearch.cluster.metadata.MetaData$Custom" -> " size = 10"
  14 = {Collections$UnmodifiableMap$UnmodifiableEntrySet$UnmodifiableEntry@14687} "interface org.elasticsearch.index.rankeval.EvaluationMetric" -> " size = 4"
  15 = {Collections$UnmodifiableMap$UnmodifiableEntrySet$UnmodifiableEntry@14688} "interface org.elasticsearch.xpack.core.indexlifecycle.LifecycleAction" -> " size = 9"
```

查询请求解析

```
public <T> Entry(Class<T> categoryClass, ParseField name, ContextParser<Object, ? extends T> parser) {
```

```
35 = {Collections$UnmodifiableMap$UnmodifiableEntrySet$UnmodifiableEntry@14757} "span_first" ->
36 = {Collections$UnmodifiableMap$UnmodifiableEntrySet$UnmodifiableEntry@14758} "percolate" ->
37 = {Collections$UnmodifiableMap$UnmodifiableEntrySet$UnmodifiableEntry@14759} "script_score" ->
38 = {Collections$UnmodifiableMap$UnmodifiableEntrySet$UnmodifiableEntry@14760} "span_term" ->
39 = {Collections$UnmodifiableMap$UnmodifiableEntrySet$UnmodifiableEntry@14761} "function_score" ->
40 = {Collections$UnmodifiableMap$UnmodifiableEntrySet$UnmodifiableEntry@14762} "geo_shape" ->
41 = {Collections$UnmodifiableMap$UnmodifiableEntrySet$UnmodifiableEntry@14763} "match" ->
42 = {Collections$UnmodifiableMap$UnmodifiableEntrySet$UnmodifiableEntry@14764} "has_parent" ->
43 = {Collections$UnmodifiableMap$UnmodifiableEntrySet$UnmodifiableEntry@14765} "span_or" ->
44 = {Collections$UnmodifiableMap$UnmodifiableEntrySet$UnmodifiableEntry@14766} "match_phrase" ->
45 = {Collections$UnmodifiableMap$UnmodifiableEntrySet$UnmodifiableEntry@14767} "exists" ->
46 = {Collections$UnmodifiableMap$UnmodifiableEntrySet$UnmodifiableEntry@14768} "has_child" ->
```

查询请求解析

SearchModule.java

```
private void registerQueryParsers(List<SearchPlugin> plugins) {
    registerQuery(new QuerySpec<>(MatchQueryBuilder.NAME, MatchQueryBuilder::new, MatchQueryBuilder::fromXContent));
    registerQuery(new QuerySpec<>(MatchPhraseQueryBuilder.NAME, MatchPhraseQueryBuilder::new, MatchPhraseQueryBuilder::fromXContent));
    registerQuery(new QuerySpec<>(MatchPhrasePrefixQueryBuilder.NAME, MatchPhrasePrefixQueryBuilder::new,
            MatchPhrasePrefixQueryBuilder::fromXContent));
    registerQuery(new QuerySpec<>(MultiMatchQueryBuilder.NAME, MultiMatchQueryBuilder::new, MultiMatchQueryBuilder::fromXContent));
    registerQuery(new QuerySpec<>(NestedQueryBuilder.NAME, NestedQueryBuilder::new, NestedQueryBuilder::fromXContent));
    registerQuery(new QuerySpec<>(DisMaxQueryBuilder.NAME, DisMaxQueryBuilder::new, DisMaxQueryBuilder::fromXContent));
    registerQuery(new QuerySpec<>(IdsQueryBuilder.NAME, IdsQueryBuilder::new, IdsQueryBuilder::fromXContent));
    registerQuery(new QuerySpec<>(MatchAllQueryBuilder.NAME, MatchAllQueryBuilder::new, MatchAllQueryBuilder::fromXContent));
    registerQuery(new QuerySpec<>(QueryStringQueryBuilder.NAME, QueryStringQueryBuilder::new, QueryStringQueryBuilder::fromXContent));
    registerQuery(new QuerySpec<>(BoostingQueryBuilder.NAME, BoostingQueryBuilder::new, BoostingQueryBuilder::fromXContent));
    BooleanQuery.setMaxClauseCount(INDICES_MAX_CLAUSE_COUNT_SETTING.get(settings));
    registerQuery(new QuerySpec<>(BoolQueryBuilder.NAME, BoolQueryBuilder::new, BoolQueryBuilder::fromXContent));
    registerQuery(new QuerySpec<>(TermQueryBuilder.NAME, TermQueryBuilder::new, TermQueryBuilder::fromXContent));
    registerQuery(new QuerySpec<>(TermsQueryBuilder.NAME, TermsQueryBuilder::new, TermsQueryBuilder::fromXContent));
```

```
private void registerQuery(QuerySpec<?> spec) {
    namedWriteables.add(new NamedWriteableRegistry.Entry(QueryBuilder.class, spec.getName().getPreferredName(), spec.getReader()));
    namedXContents.add(new NamedXContentRegistry.Entry(QueryBuilder.class, spec.getName(),
            (p, c) -> spec.getParser().fromXContent(p)));
}
```

查询请求解析

```java
/**
 * Match query is a query that analyzes the text and constructs a query as the
 result of the analysis.
 */
public class MatchQueryBuilder extends AbstractQueryBuilder<MatchQueryBuilder> {
    public static final ParseField ZERO_TERMS_QUERY_FIELD = new ParseField( name: "zero_terms_query");
    public static final ParseField CUTOFF_FREQUENCY_FIELD = new ParseField( name: "cutoff_frequency");
    public static final ParseField LENIENT_FIELD = new ParseField( name: "lenient");
    public static final ParseField FUZZY_TRANSPOSITIONS_FIELD = new ParseField( name: "fuzzy_transpositions");
    public static final ParseField FUZZY_REWRITE_FIELD = new ParseField( name: "fuzzy_rewrite");
    public static final ParseField MINIMUM_SHOULD_MATCH_FIELD = new ParseField( name: "minimum_should_match");
    public static final ParseField OPERATOR_FIELD = new ParseField( name: "operator");
    public static final ParseField MAX_EXPANSIONS_FIELD = new ParseField( name: "max_expansions");
    public static final ParseField PREFIX_LENGTH_FIELD = new ParseField( name: "prefix_length");
    public static final ParseField ANALYZER_FIELD = new ParseField( name: "analyzer");
    public static final ParseField QUERY_FIELD = new ParseField( name: "query");
    public static final ParseField GENERATE_SYNONYMS_PHRASE_QUERY = new ParseField( name: "auto_generate_synonyms_phrase_query");
```

```java
public static MatchQueryBuilder fromXContent(XContentParser parser)
```

执行文档Search请求

TransportSearchAction

- doExecute(Task, SearchRequest, ActionListener<SearchR
- shouldMinimizeRoundtrips(SearchRequest)
- ccsRemoteReduce(SearchRequest, OriginalIndices, Map<
- createSearchResponseMerger(SearchSourceBuilder, Sear
- collectSearchShards(IndicesOptions, String, String, Atom
- processRemoteShards(Map<String, ClusterSearchShards
- buildConnectionLookup(String, Function<String, Discove
- mergeShardsIterators(GroupShardsIterator<ShardIterato

TransportSearchAction::doExecute

获取远程集群合待搜索索引的对应关系

```
final Map<String, List<String>> groupedIndices = groupClusterIndices(getRemoteClusterNames(), indices, indexExists);
```

若无远程集群(跨集群搜索)，则只执行本地集群搜索

```
if (remoteClusterIndices.isEmpty()) {
    executeLocalSearch(task, timeProvider, searchRequest, localIndices, clusterState, listener);
```

TransportSearchAction::executeLocalSearch

TransportSearchAction::executeSearch

跨集群搜索最终也会调回的到此方法

检测集群有无全局读阻塞

```
clusterState.blocks().globalBlockedRaiseException(ClusterBlockLevel.READ);
```

获取真实索引名(concreteIndex)

```
final Index[] indices = resolveLocalIndices(localIndices, searchRequest,
concreteIndices[i] = indices[i].getName();
```

若无远程集群(跨集群搜索), 远程集群分片为空

```
Collections.emptyList(),
```

合并待搜索本地集群分片和远程集群分片

```
GroupShardsIterator<ShardIterator> localShardsIterator = clusterService.operationRouting().searchShards(clusterState,
        concreteIndices, routingMap, searchRequest.preference(), searchService.getResponseCollectorService(), nodeSearchCounts);
GroupShardsIterator<SearchShardIterator> shardIterators = mergeShardsIterators(localShardsIterator, localIndices,
    searchRequest.getLocalClusterAlias(), remoteShardIterators);
```

执行文档Search请求

TransportSearchAction

- doExecute(Task, SearchRequest, ActionLi
- shouldMinimizeRoundtrips(SearchReque
- ccsRemoteReduce(SearchRequest, Origin
- createSearchResponseMerger(SearchSourceBuilder, Sear
- collectSearchShards(IndicesOptions, String, String, Atom
- processRemoteShards(Map<String, ClusterSearchShards
- buildConnectionLookup(String, Function<String, Discove
- mergeShardsIterators(GroupShardsIterator<ShardIterato

TransportSearchAction::executeSearch

如果只有一个待搜索分片，搜索类型强制改为QUERY_THEN_FETCH

```
// optimize search type for cases where there is only one shard group to search on
if (shardIterators.size() == 1) {
    // if we only have one group, then we always want Q_T_F, no need for DFS, and no need to do THEN since we hit one shard
    searchRequest.searchType(QUERY_THEN_FETCH);
}
```

如果只需查询建议，搜索类型强制改为QUERY_THEN_FETCH

TransportSearchAction::searchAsyncAction

TransportSearchAction::searchAsyncAction

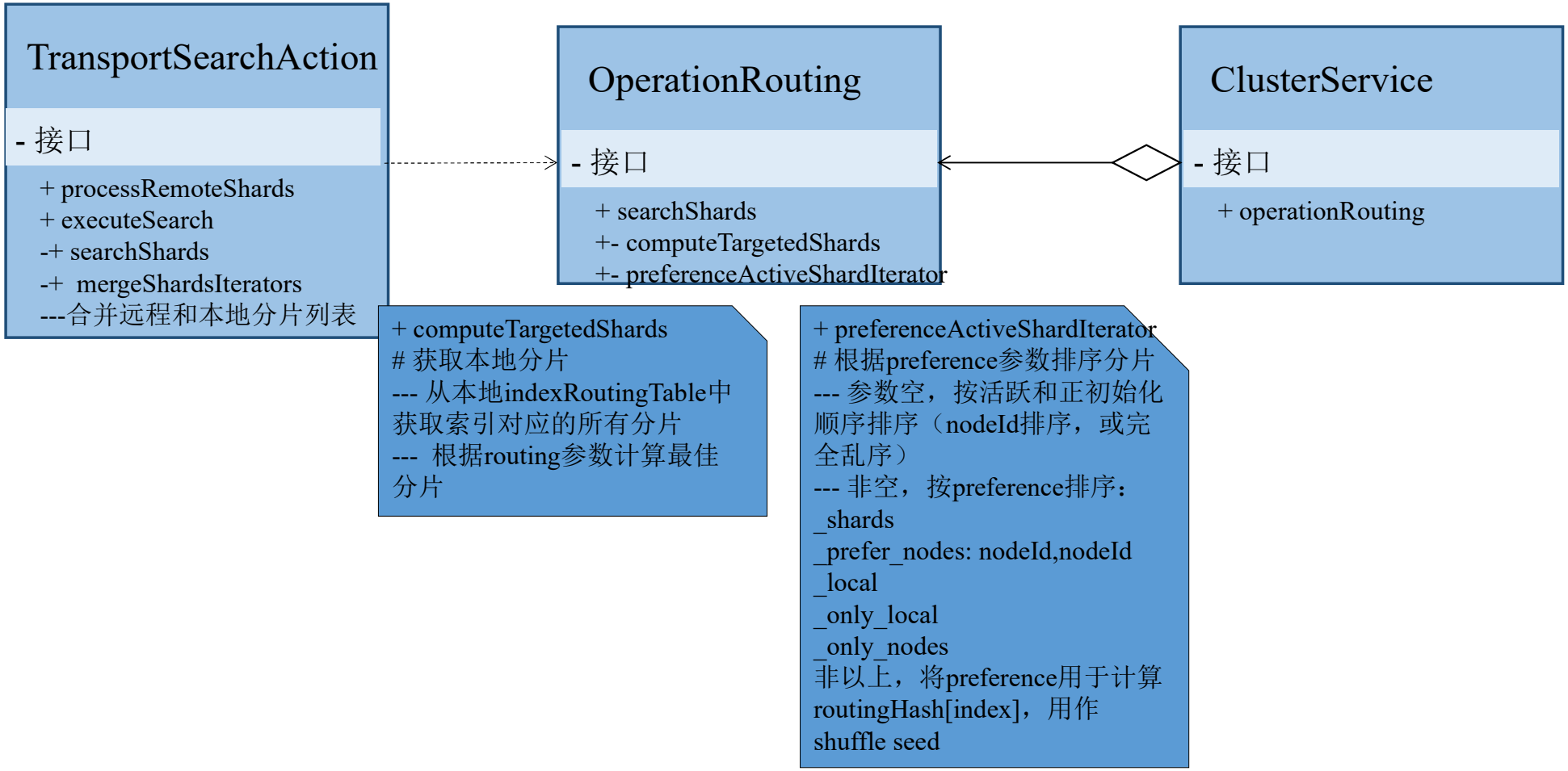
获取
DFS_QUERY_THEN_FETCH Action 或
QUERY_THEN_FETCH Action

```
switch (searchRequest.searchType()) {
    case DFS_QUERY_THEN_FETCH:
        searchAsyncAction = new SearchDfsQueryThenFetchAsyncAction(logger, searchTransportService, connectionLookup,
            aliasFilter, concreteIndexBoosts, indexRoutings, searchPhaseController, executor, searchRequest, listener,
            shardIterators, timeProvider, clusterStateVersion, task, clusters);
        break;
    case QUERY_THEN_FETCH:
        searchAsyncAction = new SearchQueryThenFetchAsyncAction(logger, searchTransportService, connectionLookup,
            aliasFilter, concreteIndexBoosts, indexRoutings, searchPhaseController, executor, searchRequest, listener,
            shardIterators, timeProvider, clusterStateVersion, task, clusters);
        break;
```

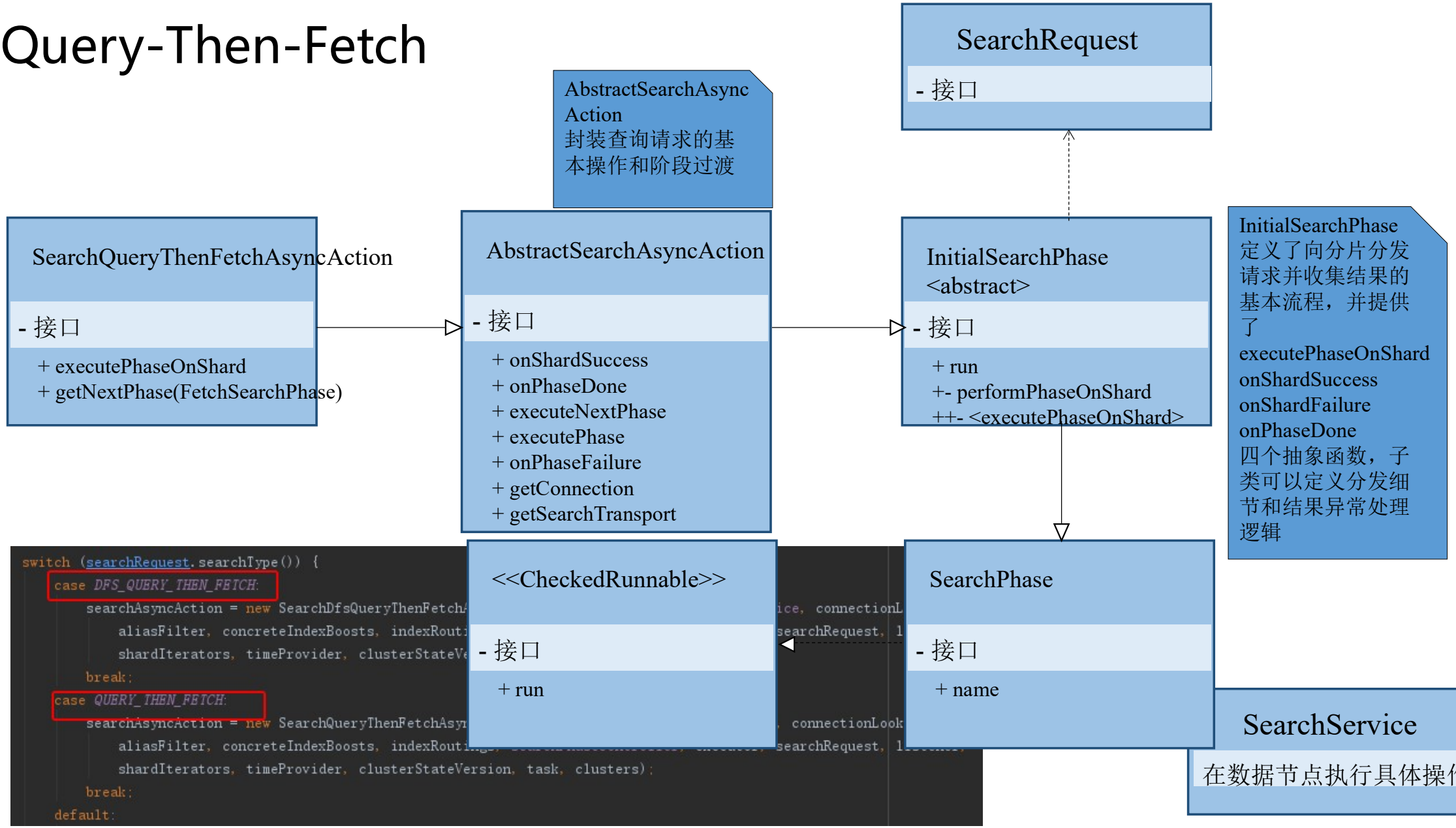
```
searchAsyncAction().start();
```

启动对应的Action'

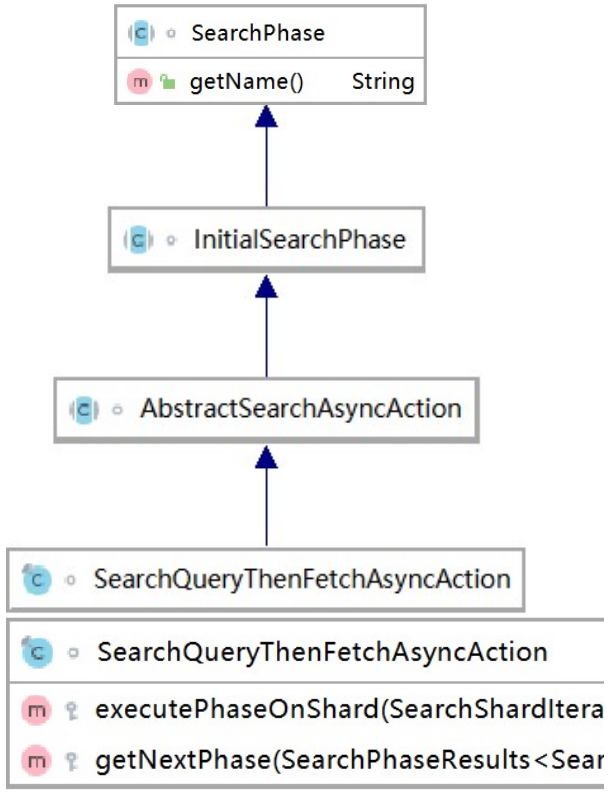
构造目标Shard列表



Query-Then-Fetch



Query-Then-Fetch: Query



AbstractSearchAsyncAction::start

AbstractSearchAsyncAction::executePhase

InitialSearchPhase::run

遍历每个待搜索分片

InitialSearchPhase::performPhaseOnShard

SearchQueryThenFetchAsyncAction::executePhaseOnShard

```
private void performPhaseOnShard(final int shardIndex, final SearchShardIterator shardIt, final ShardRouting shard)
    executePhaseOnShard(shardIt, shard,
        new SearchActionListener<FirstResult>(shardIt.newSearchShardTarget(shard.currentNodeId()), shardIndex) {...});
```

SearchActionListener::onResponse Query阶段搜索返回后

InitialSearchPhase::onShardResult 执行下一阶段Fetch

InitialSearchPhase::successfulShardExecution

AbstractSearchAsyncAction::onPhaseDone

AbstractSearchAsyncAction::executeNextPhase

```
abstract class AbstractSearchAsyncAction  private void executePhase(SearchPhase phase)
    phase.run();
```

SearchQueryThenFetchAsyncAction::executePhaseOnShard

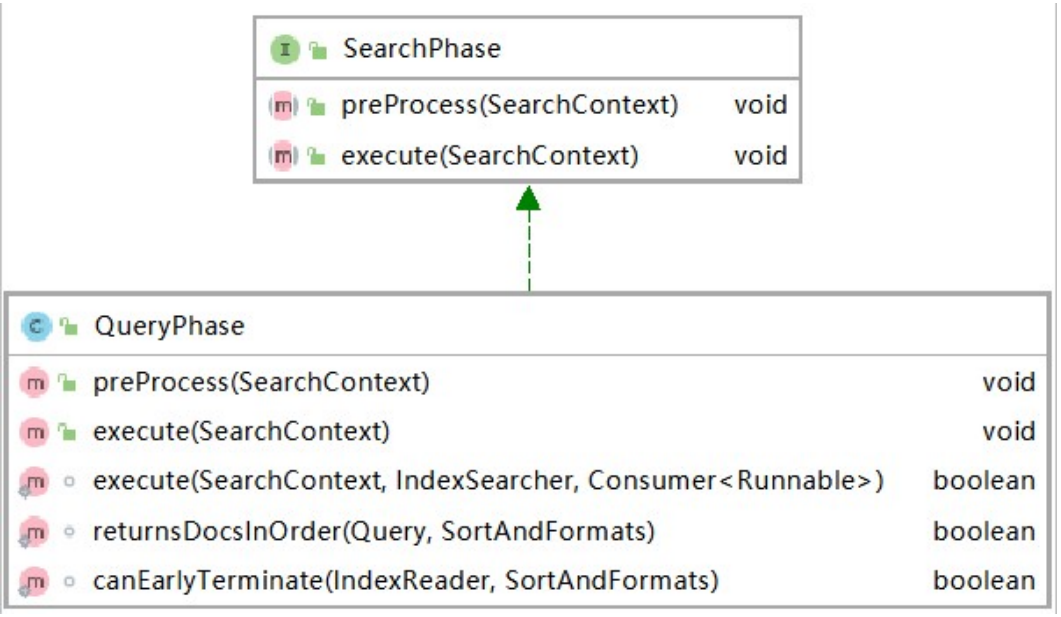
Rpc调用，在本地或远程节点执行

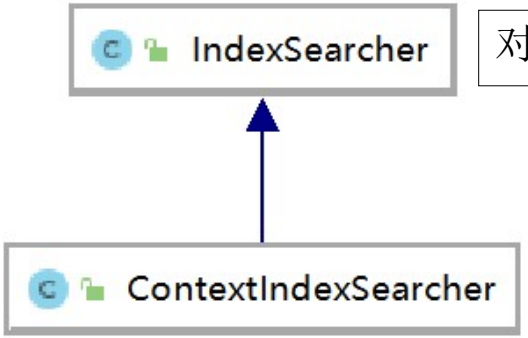
SearchTransportService::sendExecuteQuery

在具体的节点执行QueryPhase::execute

SearchService::executeQueryPhase

Query-Then-Fetch: Query





对Lucene查询的封装

reader	IndexReader
readerContext	IndexReaderContext
leafContexts	List<LeafReaderContext>
leafSlices	LeafSlice[]

QueryPhase::execute
--- 执行并得到搜索结果(TopDocs 文档Id)

DefaultSearchContext.searcher() 获取 ContextIndexSearcher

QueryPhase.static execute

ContextIndexSearcher::search

```
public class QueryPhase
public void execute(SearchContext searchContext)
static boolean execute(SearchContext searchContext,
searcher.search(query, queryCollector);
```

处理查询中的suggest

处理查询中的聚合

```
boolean rescore = execute(searchContext, searchContext.searcher(),

if (rescore) { // only if we do a regular search
    rescorePhase.execute(searchContext);
}
suggestPhase.execute(searchContext);
aggregationPhase.execute(searchContext);
```

IndexSearcher	
count(Query)	int
getSlices()	LeafSlice[]
searchAfter(ScoreDoc, Query, int)	TopDocs
search(Query, int)	TopDocs
search(Query, Collector)	void
search(Query, int, Sort, boolean)	TopFieldDocs

InitialSearchPhase

- 作用

  + 定义了向分片分发请求并收集结果的基本流程

- run

  + 检测是否有分片缺失（allowPartialResult==false）
  ---allowPartialResult==true的情况允许返回部分结果,无需检测分片缺失
  + 在每个分片上执行performPhaseOnShard，发送请求

- performPhaseOnShard

  + PendingExecutions.tryRun
  -+ <executePhaseOnShard> 发送查询请求
  --+onShardResult 调用<onShardSuccess>合并查询结果。统计所有分片是否都处理完，调用<onPhaseDone>，进入Fetch阶段

- <executePhaseOnShard>

  + 子类自定义每个分片的处理逻辑

PendingExecutions

- 接口

  + tryRun
  +- tryQueue 用于控制每个节点的并发请求数，由maxConcurrentRequestsPerNode 控制，超出部分存放在队列中，依次取出执行

AbstractSearchAsyncAction

- 作用

+ 实现向分片分发请求并收集结果的基本操作
+ 实现了不同查询阶段切换的逻辑

- onShardSuccess

+ QueryPhaseResultConsumer::consumeResult 合并结果
-+ consumeInternal(QuerySearchResult)
--+ InternalAggregations::reduce(InternalAggregations[])
---- 聚合查询结果合并
--+ 文档查询结果TopDocs[]合并，得到topN结果
---+ TopDocs::merge (Lucene)
--+ 赋值null，释放空间(QuerySearchResult)

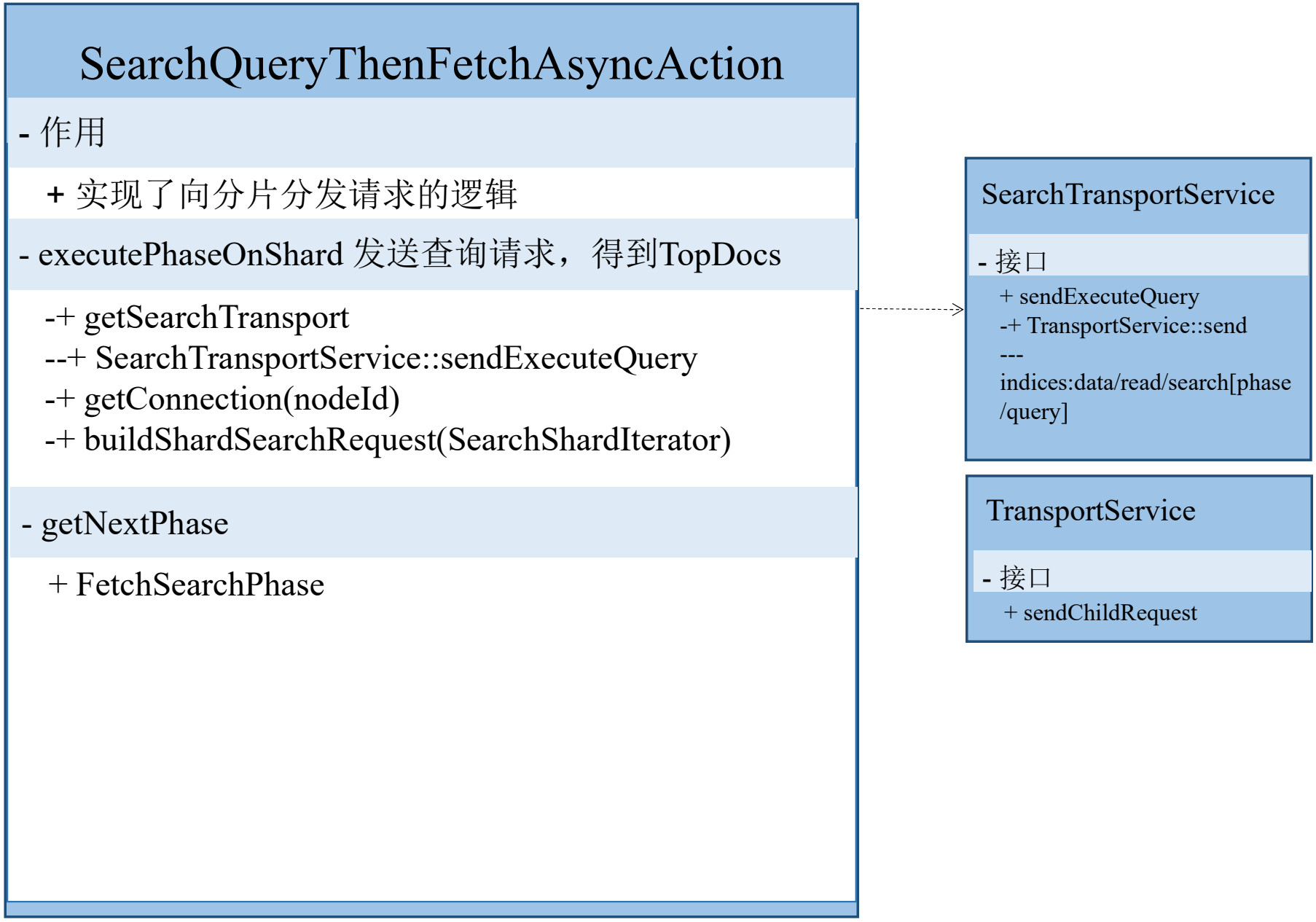
- onPhaseDone 切换至下一查询阶段(Fetch)

-+ <getNextPhase>
-+ executeNextPhase
--+ 若不允许返回部分结果(allowPartialResults),且存在分片查询异常，查询失败
--+ executePhase(nextPhase) 执行下一阶段

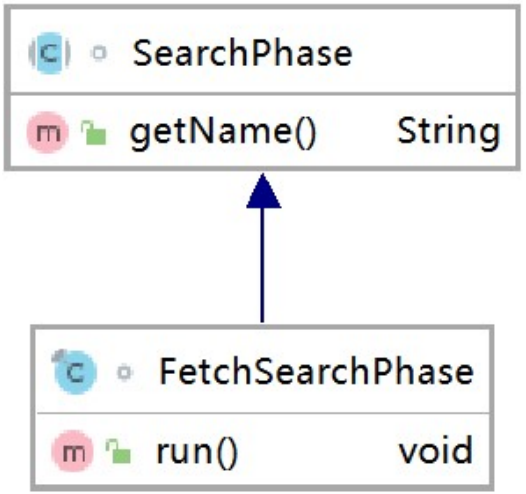
SearchPhaseController
参与查询结果合并

- 接口

+ reduceContextFunction
+ mergeTopDocs


SearchQueryThenFetchAsyncAction
- 作用
+ 实现了向分片分发请求的逻辑
- executePhaseOnShard 发送查询请求，得到TopDocs
-+ getSearchTransport
--+ SearchTransportService::sendExecuteQuery
-+ getConnection(nodeId)
-+ buildShardSearchRequest(SearchShardIterator)
- getNextPhase
+ FetchSearchPhase
SearchTransportService
- 接口
+ sendExecuteQuery
-+ TransportService::send
---
indices:data/read/search[phase/query]
TransportService
- 接口
+ sendChildRequest

Query-Then-Fetch: Query



FetchPhase::run
--- 根据Query阶段得到的TopDocs，获取文档内容

FetchPhase::innerRun

合并从各分片接收到的TopDocs

```
private void innerRun()
SearchPhaseController.ReducedQueryPhase reducedQueryPhase = resultConsumer.reduce();
```

从各分片取回文档

```
for (int i = 0; i < docIdsToLoad.length; i++) {
    IntArrayList entry = docIdsToLoad[i];
    SearchPhaseResult queryResult = queryResults.get(i);
    if (entry == null) {...} else {
        SearchShardTarget searchShardTarget = queryResult.getSearchShardTarget();
        Transport.Connection connection = context.getConnection(searchShardTarget.getClusterAlias(),
            searchShardTarget.getNodeId());
        ShardFetchSearchRequest fetchSearchRequest = createFetchRequest(queryResult.queryResult().getRequ
            lastEmittedDocPerShard, searchShardTarget.getOriginalIndices());
        executeFetch(i, searchShardTarget, counter, fetchSearchRequest, queryResult.queryResult(),
            connection);
    }
}
```

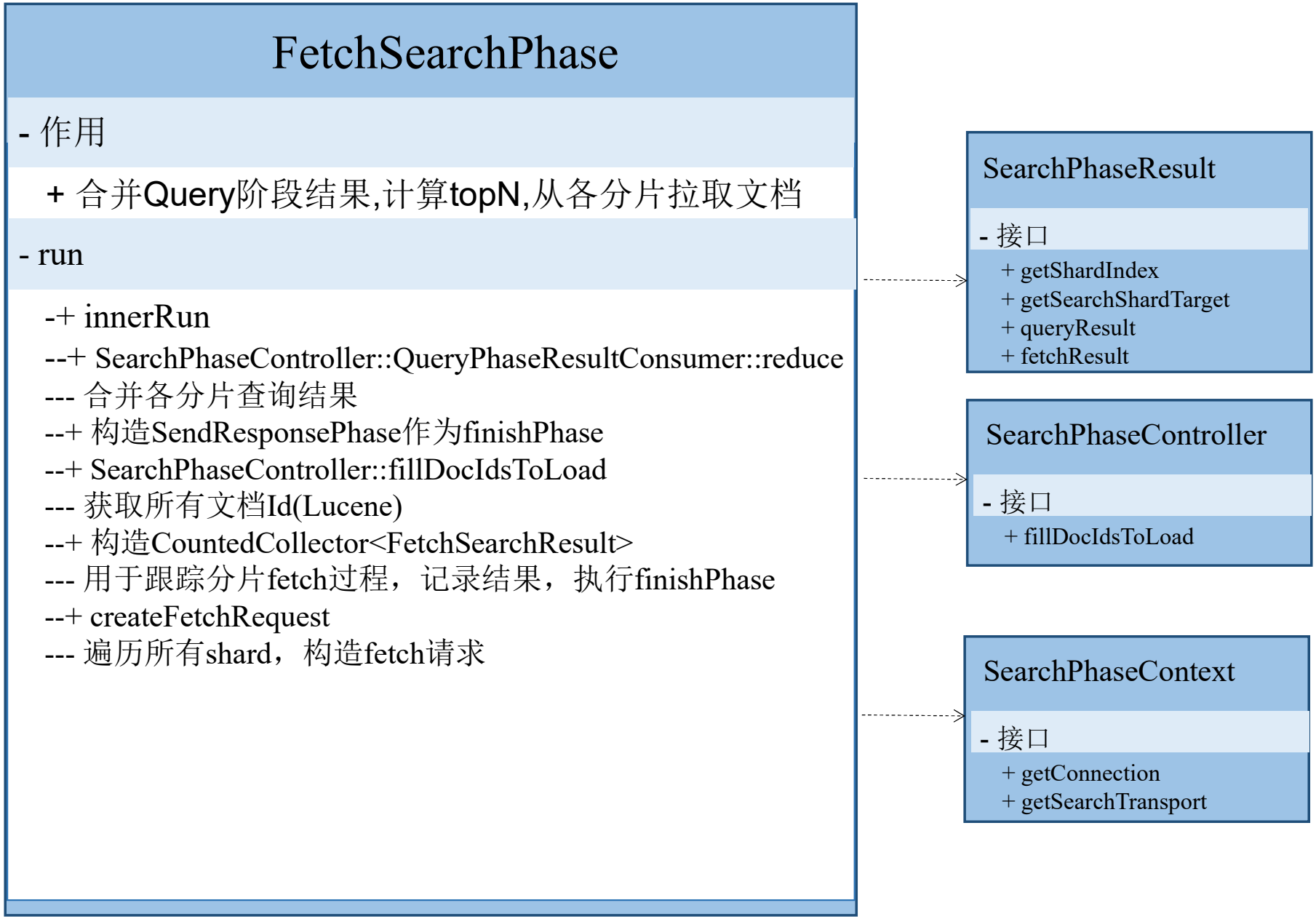
SearchTransportService::sendExecuteFetch

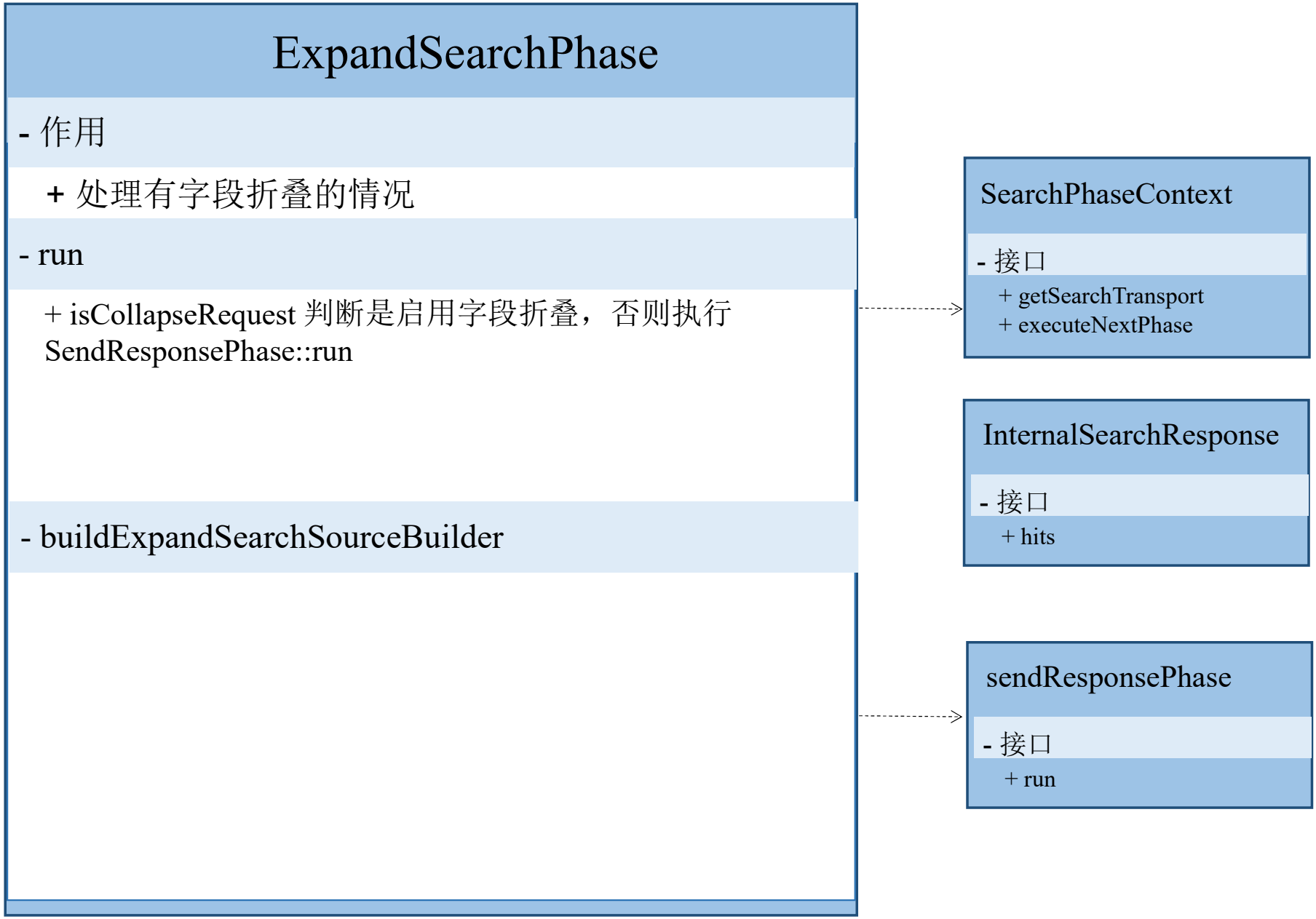
Rpc调用，在本地或远程节点执行

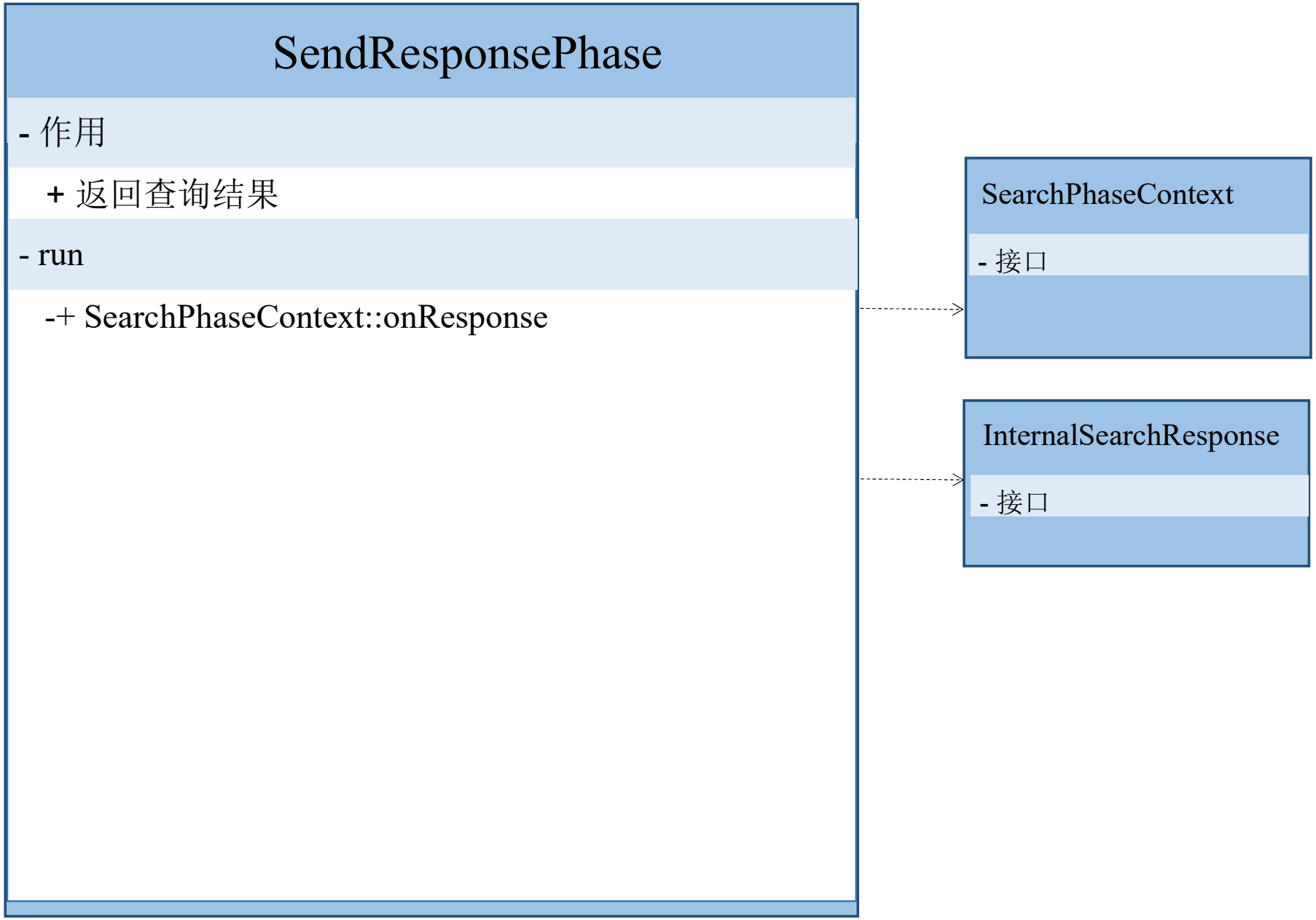
SearchService::executeFetchPhase

FetchPhase::execute

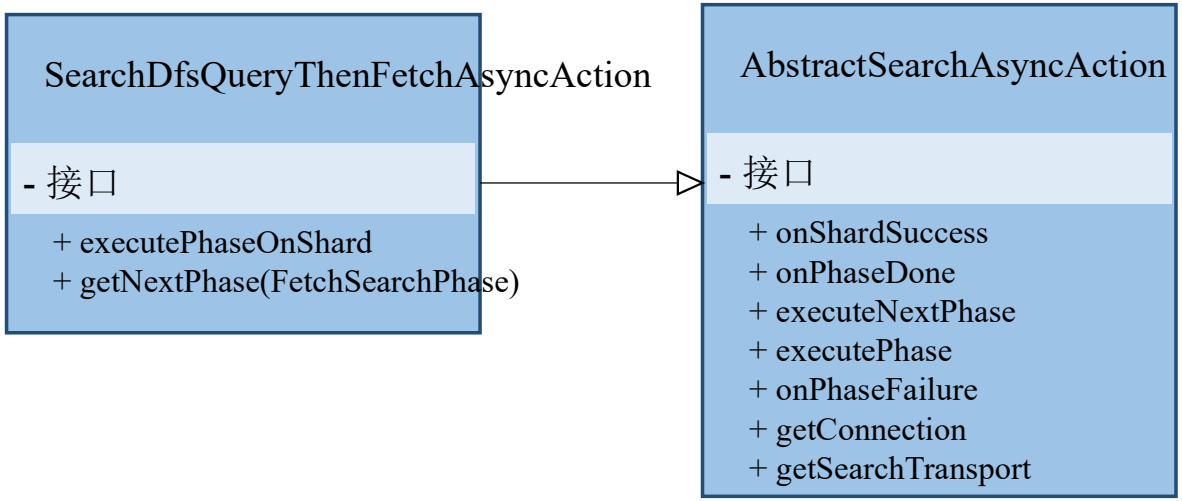
FetchPhase::createSearchHit

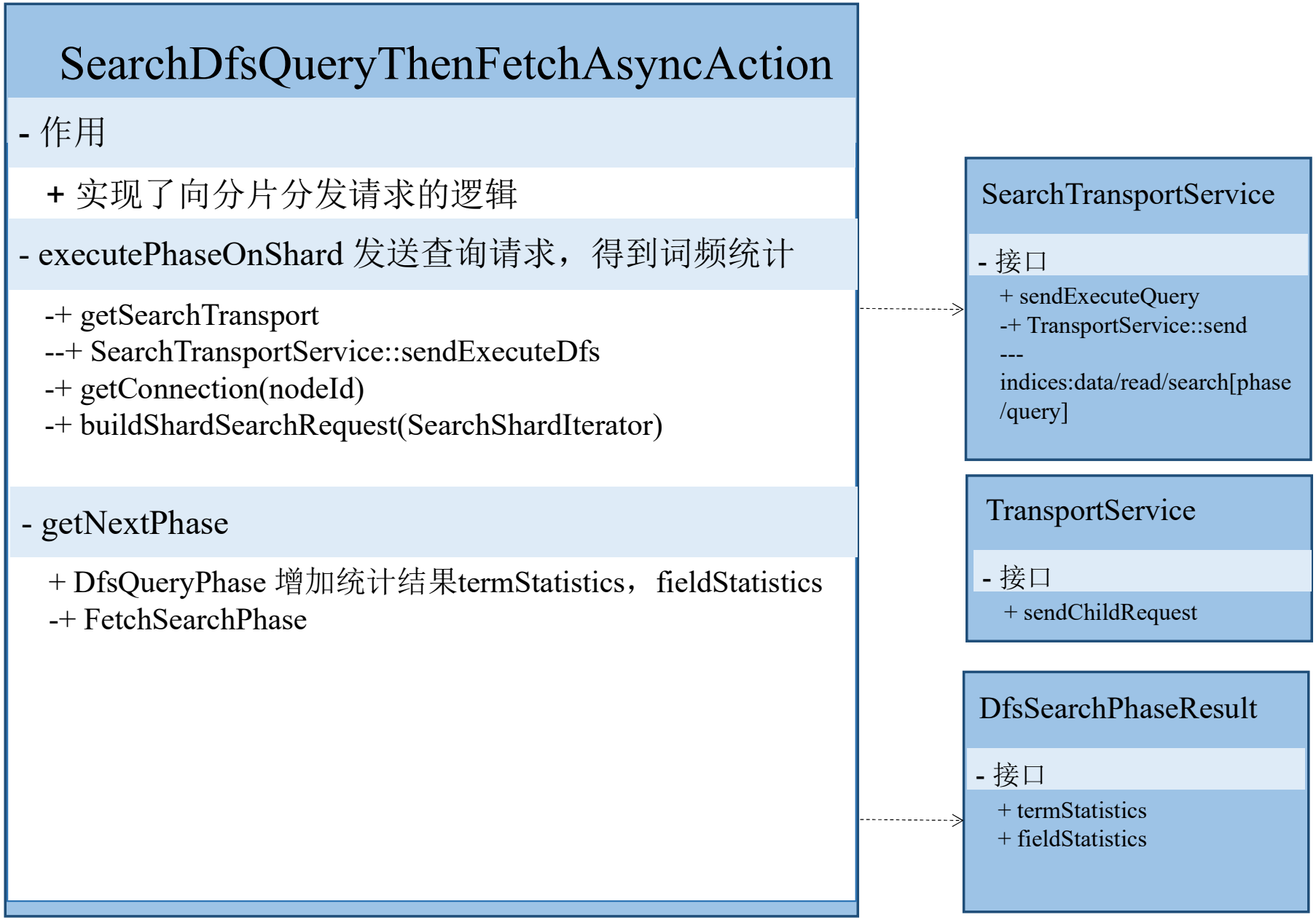

FetchSearchPhase
- 作用
+ 合并Query阶段结果,计算topN,从各分片拉取文档
- run
-+ innerRun
--+ SearchPhaseController::QueryPhaseResultConsumer::reduce
--- 合并各分片查询结果
--+ 构造SendResponsePhase作为finishPhase
--+ SearchPhaseController::fillDocIdsToLoad
--- 获取所有文档Id(Lucene)
--+ 构造CountedCollector<FetchSearchResult>
--- 用于跟踪分片fetch过程，记录结果，执行finishPhase
--+ createFetchRequest
--- 遍历所有shard，构造fetch请求
SearchPhaseResult
- 接口
+ getShardIndex
+ getSearchShardTarget
+ queryResult
+ fetchResult
SearchPhaseController
- 接口
+ fillDocIdsToLoad
SearchPhaseContext
- 接口
+ getConnection
+ getSearchTransport

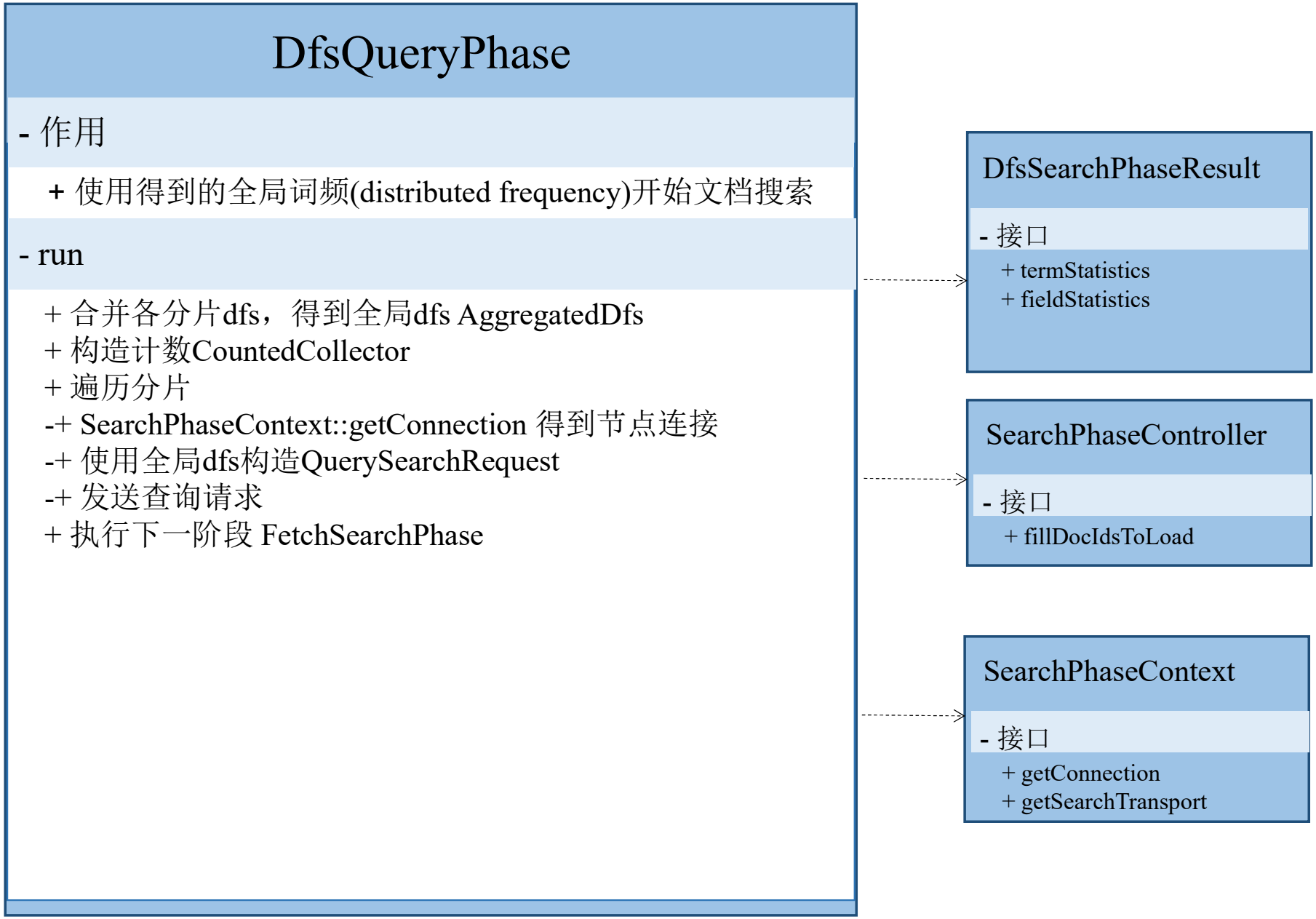

ExpandSearchPhase
- 作用
+ 处理有字段折叠的情况
- run
+ isCollapseRequest 判断是启用字段折叠，否则执行 SendResponsePhase::run
- buildExpandSearchSourceBuilder
SearchPhaseContext
- 接口
+ getSearchTransport
+ executeNextPhase
InternalSearchResponse
- 接口
+ hits
sendResponsePhase
- 接口
+ run


SendResponsePhase
- 作用
+ 返回查询结果
- run
-+ SearchPhaseContext::onResponse
SearchPhaseContext
- 接口
InternalSearchResponse
- 接口

Dfs-Query-Then-Fetch




SearchDfsQueryThenFetchAsyncAction
- 作用
+ 实现了向分片分发请求的逻辑
- executePhaseOnShard 发送查询请求，得到词频统计
-+ getSearchTransport
--+ SearchTransportService::sendExecuteDfs
-+ getConnection(nodeId)
-+ buildShardSearchRequest(SearchShardIterator)
- getNextPhase
+ DfsQueryPhase 增加统计结果termStatistics，fieldStatistics
-+ FetchSearchPhase
SearchTransportService
- 接口
+ sendExecuteQuery
-+ TransportService::send
---
indices:data/read/search[phase/query]
TransportService
- 接口
+ sendChildRequest
DfsSearchPhaseResult
- 接口
+ termStatistics
+ fieldStatistics

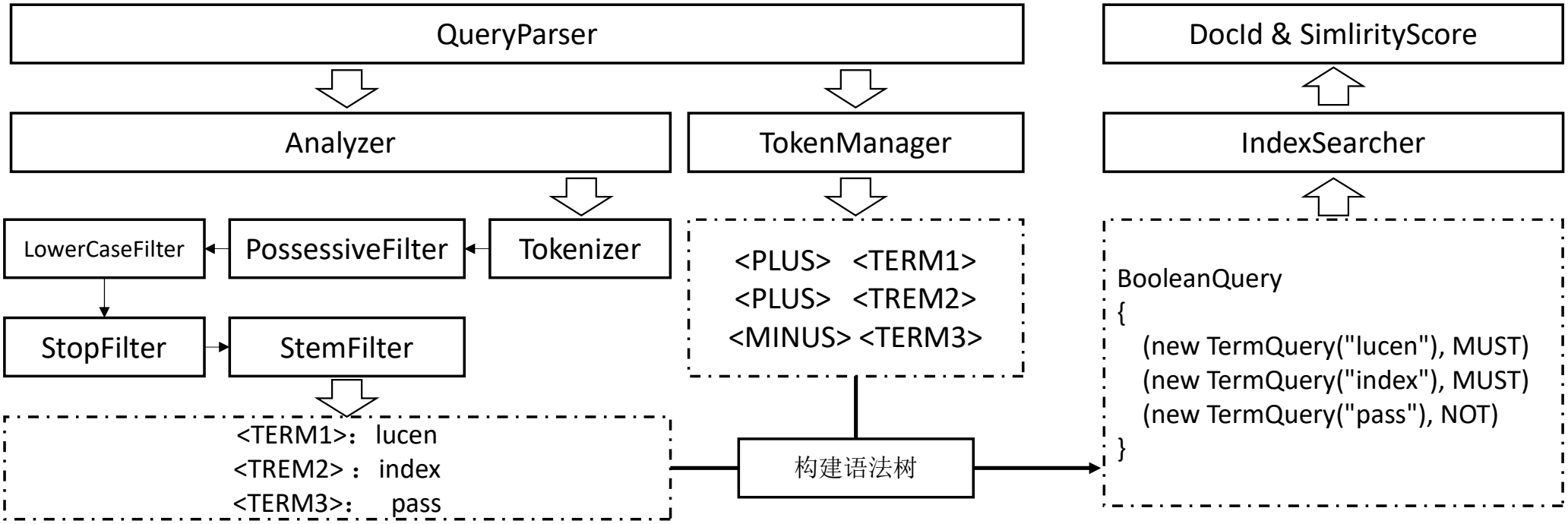

DfsQueryPhase
- 作用
+ 使用得到的全局词频(distributed frequency)开始文档搜索
- run
+ 合并各分片dfs，得到全局dfs AggregatedDfs
+ 构造计数CountedCollector
+ 遍历分片
-+ SearchPhaseContext::getConnection 得到节点连接
-+ 使用全局dfs构造QuerySearchRequest
-+ 发送查询请求
+ 执行下一阶段 FetchSearchPhase
DfsSearchPhaseResult
- 接口
+ termStatistics
+ fieldStatistics
SearchPhaseController
- 接口
+ fillDocIdsToLoad
SearchPhaseContext
- 接口
+ getConnection
+ getSearchTransport

Lucene搜索

```
[0] Lucene is a high performance text search/index engine  ->  lucen high perform text search index engin
[1] Lucene is good at information retrieval                ->  lucen good inform retriev
[2] Plain text passed to Lucene for indexing               ->  plain text pass lucen index
```

- 需要检索出包含Lucene与index，但是不包含passed的文档

构造查询语句，+Lucene +index -passed，或者 Lucene AND index NOT passed



Lucene索引结构

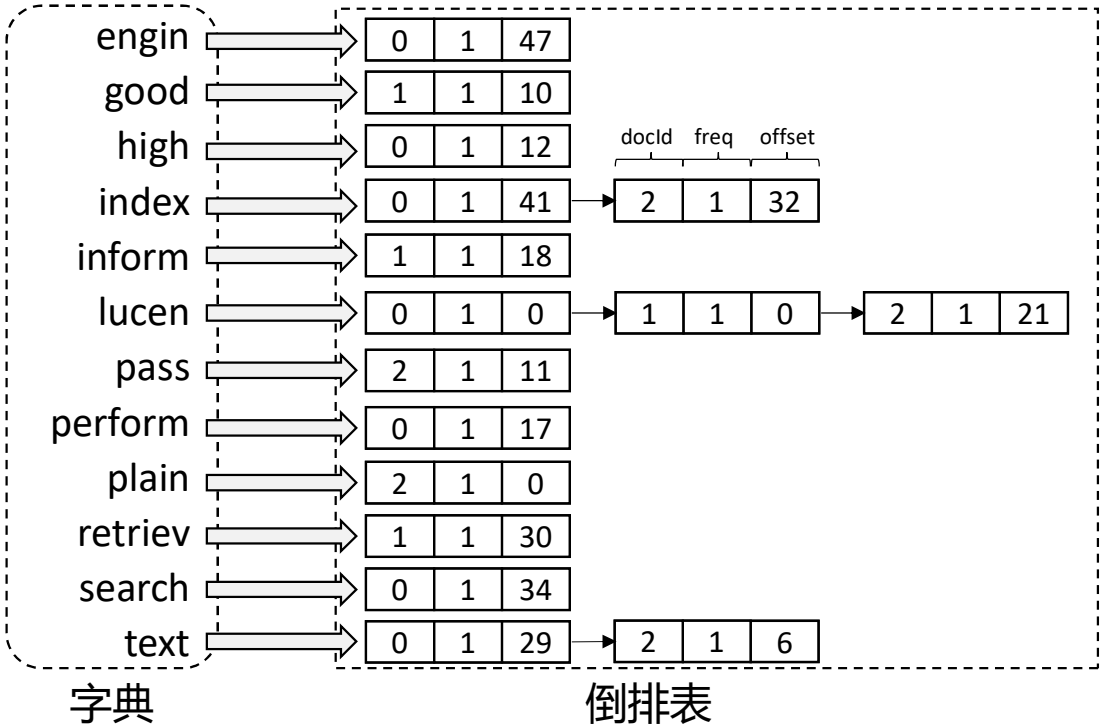
- [0] Lucene is a high performance text search/index engine -> lucen high perform text search index engin
- [1] Lucene is good at information retrieval -> lucen good inform retriev
- [2] Plain text passed to Lucene for indexing -> plain text pass lucen index

1、文档分词
2、单词小写化（英文）
3、去除标点和停用词
4、单词转为词根形式

inf	**info**	**inform**	**ear**	**earlier**

字典压缩

inf	**[3]o**	**[4]rm**	**ear**	**[3]lier**

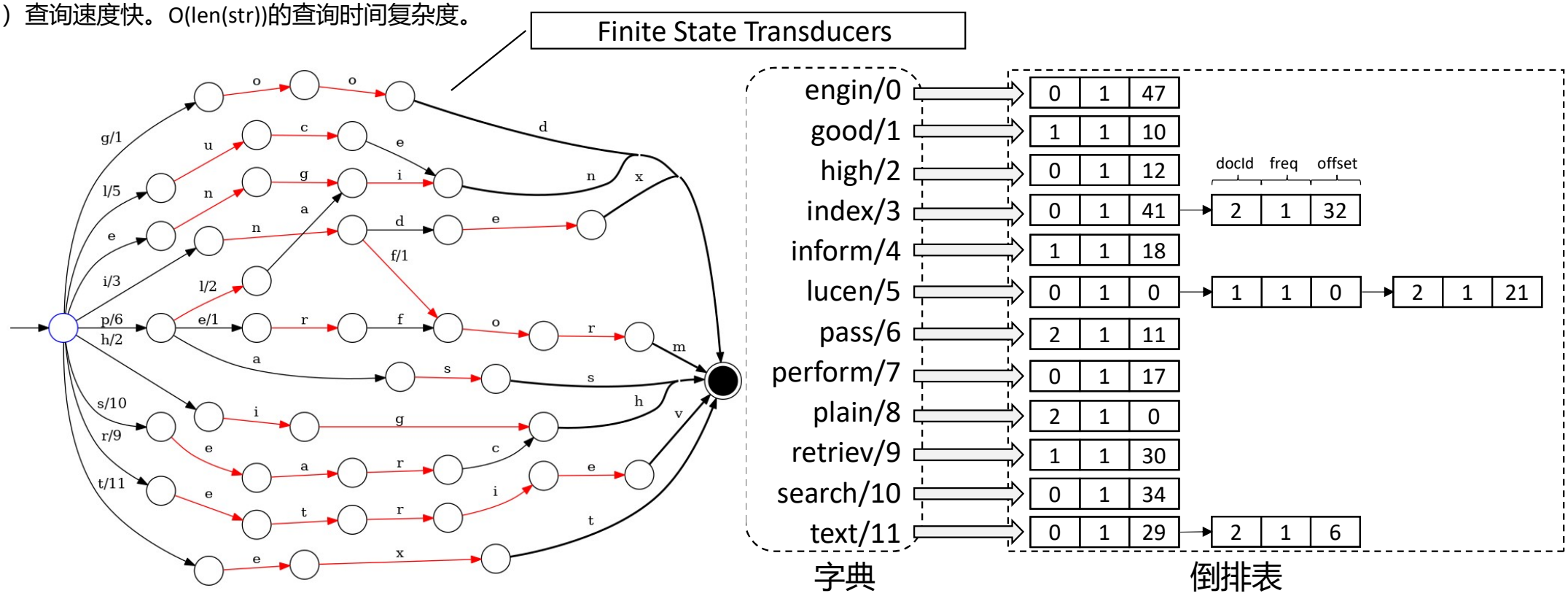


Lucene词典

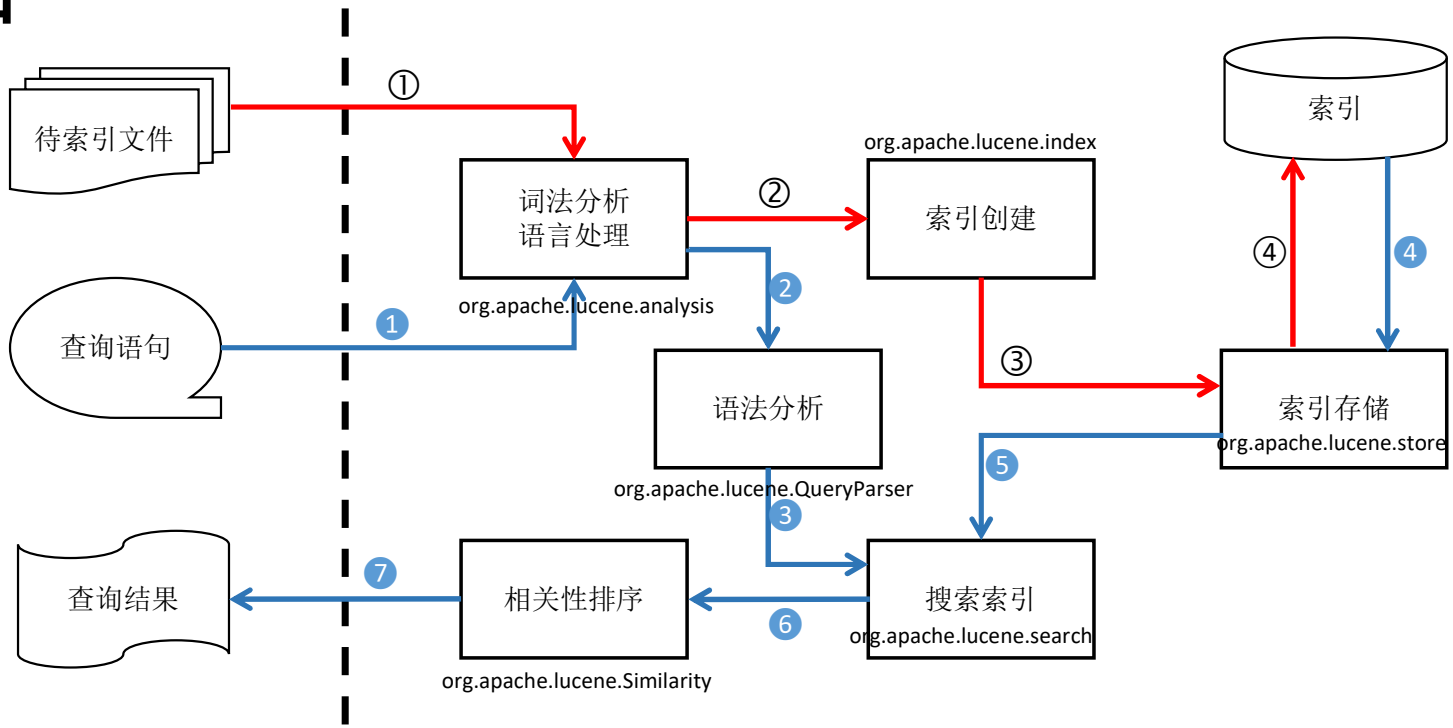
FST(有限状态转换器)优势：
1）空间占用小。通过对词典中单词前缀和后缀的重复利用，压缩了存储空间
2）查询速度快。O(len(str))的查询时间复杂度。

如何快速搜索字典，定位倒排表

列表？哈希表？跳表？B-Tree？



Lucene索引与检索—总结



- 待索引文件经过语法分析和语言处理形成一系列词（Term）
- Lucene根据Term创建词典和倒排索引表
- 通过索引存储模块将索引存入磁盘

查询语句经过语法分析和语言处理形成一系列词（Term）

通过语法分析得到一个查询树

将索引从磁盘读入内存

利用查询树构造出查询条件

通过查询条件得到多个文档链表，并进行交集、差集、并集运算，得到结果文档

根据查询语句与文档内容，使用VSM模型对文档相关性进行打分并排序