# 《中间件技术课程报告》

姓名：魏伊雪

学号：2017104156

项目地址：https://github.com/58code/Oceanus

## 58同城数据库中间件oceanus研究报告

### 一、背景分析

随着互联网产品在体量和规模上日益膨胀，无论是Oracle还是MySQL，都会第一时间面临来自磁盘，CPU和内存等单机瓶颈，为此，产品方除了需要不断购买成本难以控制的高规格服务器，还要面临不断迭代的在线数据迁移。在这种情况下，无论是海量的结构化数据还是快速成长的业务规模，都迫切需要一种水平扩展的方法将存储成本分摊到成本可控的商用服务器上。同时，也希望通过线性扩容降低全量数据迁移对线上服务带来的影响，分库分表方案便应运而生。

在DB存储需求中，尽管业务不同，技术难点还是类似的，开源世界有很多DB中间件，解决方案也以通用方案为主，满足业务需要为前提，支持各种类型的需求。中间件技术可以解决的数据库问题有： 简化开发，降低分库分表复杂度、分库分表透明化、统一配置管理、提供监控、链接管理、HA支持、LoadBlance支持、提供更好的扩展能力。

### 二、Oceanus介绍

Oceanus致力于打造一个功能简单、可依赖、易于上手、易于扩展、易于集成的解决方案，甚至是平台化系统。拥抱开源，提供各类插件机制集成其他开源项目，新手可以在几分钟内上手编程，分库分表逻辑不再与业务紧密耦合，扩容有标准模式，减少意外错误的发生。

数据在单据数据库存储，与分库分表场景的多库存储，在使用方法上有很

大的不同。各类数据库中间件产品极力简化使用者的开发成本，希望做到

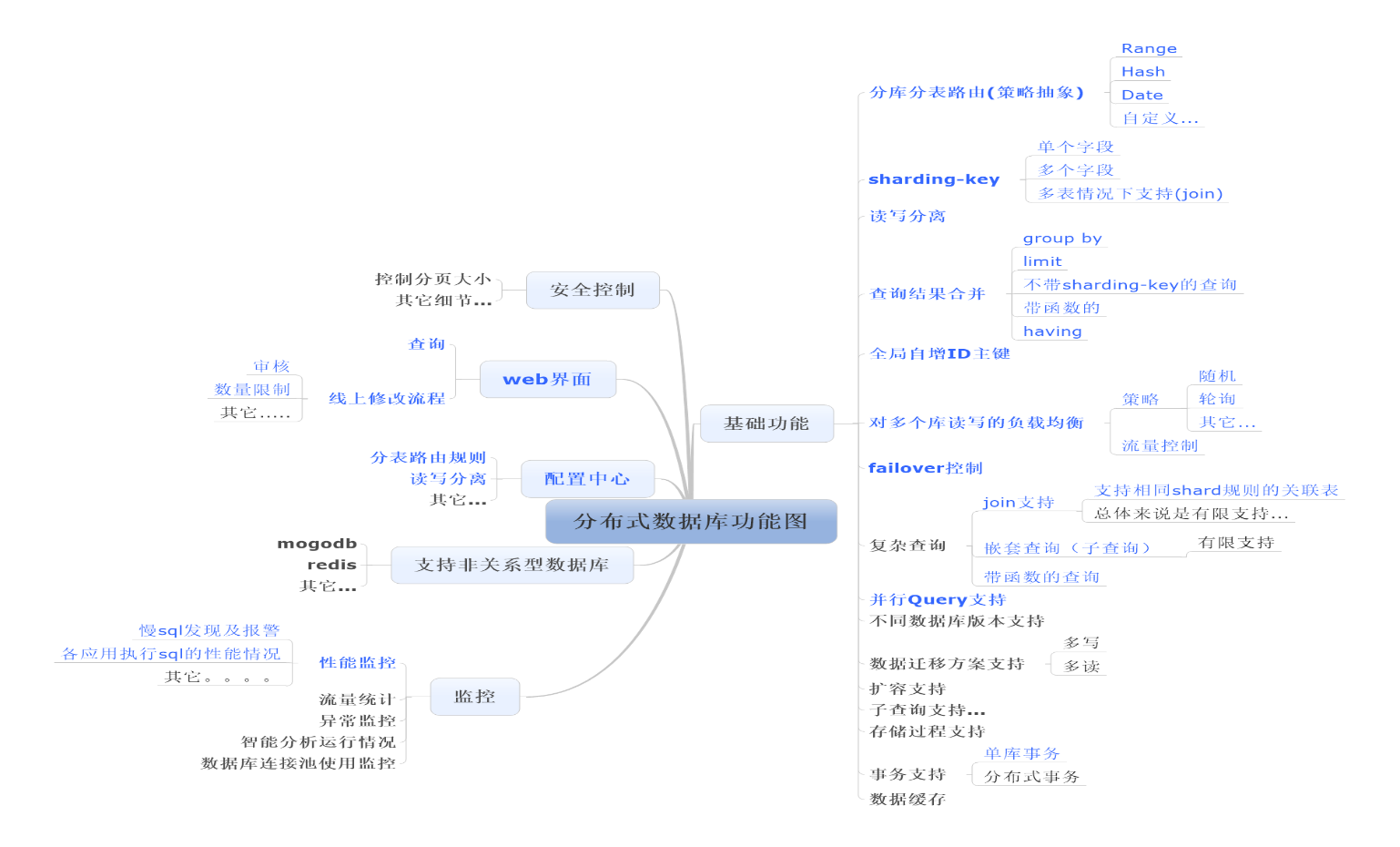
操作单库场景一样操作多库。

Oceanus 通过标准配置，与业务方制定好路由规则，程序运行时提交简单 sql 即可自动路由到不同库中执行，合并结果后返回上层。

### 三、功能及技术分析

实现方法： 应用硬编码实现、DAL封装（或者ORM封装）、面向JDBC标准、基于Mysql客户端协议。

设计原则：模块化、职责分离、支持封装和扩展



功能规划图

总体思路：通过约定的XML规则（分表分库规则）和 封装jdbc的Connection和PreparedStatement来实现SQL解析，sql路由和sql重写。

### 四、源码分析

**3个核心类：**

ConnectionWrapper（JDBC Connection包装）PreparedStatementWrapper( JDBC PreparedStatement包装)

SimpleExecutor(sql执行器，类似mybatis的SimpleExecutor)

**3个上下文传参数：**

ConnectionContext，StatementContext，transactionContext

**真正干活的类：**

1. DefaultStatementContextBuilder类：

解析SQL并保存BatchItem到StatementContext

**Java代码  [收藏代码](javascript:void())**

1. **public** StatementContext build(String sql, StatementContext context)
2. **throws** SQLException {
3. **if** (context == ) {
4. context = **new** StatementContext();
5. StatementContext.setContext(context);
6. **if** (logger.isDebugEnabled()) {
7. logger.debug("create context!sql=" + sql);
8. }
9. }
10. **if** (context.getCurrentBatch().getSql() == ) {
11. context.getCurrentBatch().setSql(sql);
12. }
14. StatementContextHandler handler = ;
15. **if** (context.isBatch()) {
16. handler = HandlerFactory.create(StatementType.BATCH);
17. StatementContext resultContext = handler.handle(sql, context);
18. processPreparedValues(resultContext);
19. **return** resultContext;
20. }
22. TrackerExecutor.trackBegin(TrackPoint.PARSE\_SQL, sql);
23. SQLParser parser = StatementHelper.createSQLParser();
24. **try** {
25. DMLStatementNode statementNode = (DMLStatementNode) parser
26. .parseStatement(sql);
27. **switch** (statementNode.getNodeType()) {
28. **case** NodeTypes.CURSOR\_NODE:
29. handler = HandlerFactory.create(StatementType.SELECT);
30. **break**;
31. **case** NodeTypes.DELETE\_NODE:
32. handler = HandlerFactory.create(StatementType.DELETE);
33. **break**;
34. **case** NodeTypes.UPDATE\_NODE:
35. handler = HandlerFactory.create(StatementType.UPDATE);
36. **break**;
37. **case** NodeTypes.INSERT\_NODE:
38. handler = HandlerFactory.create(StatementType.INSERT);
39. **break**;
40. **case** NodeTypes.CALL\_STATEMENT\_NODE:
41. handler = HandlerFactory.create(StatementType.CALLABLE);
42. **break**;
43. }
45. StatementContext resultContext = handler.handle(statementNode,
46. context);
48. TrackerExecutor.trackEnd(TrackPoint.PARSE\_SQL);
50. processPreparedValues(resultContext);
51. **return** resultContext;
53. } **catch** (StandardException se) {
54. System.out.println("sql parse error, sql:"+sql);
55. se.printStackTrace();
56. } **catch** (Exception e) {
57. e.printStackTrace();
58. }
59. processPreparedValues(context);
60. **return** context;
61. }
62. DefaultTargetDispatcher：根据batchItem和batchItem中的tableInfo获取路由信息和重写SQL，达到路由到指定的分库分表的目的

**Java代码  [收藏代码](javascript:void())**

1. Set<RouteTarget> getSpecifyTargets(TableInfo tableInfo, BatchItem batchItem) {
2. Set<RouteTarget> targetSet = **new** LinkedHashSet<RouteTarget>();
3. Configurations configurations = Configurations.getInstance();
5. /\*\*
6. \* 解析where中符合分库分表字段的值
7. \*/
8. Map<String, List<TableColumn>> resolveColumns = RouteHelper
9. .getResolveColumns(tableInfo.getOrgName(),
10. batchItem.getAnalyzeResult());
11. List<Map<String, Object>> parameters = RouteHelper
12. .getParameterValues(resolveColumns);
13. Set<Integer> indexs = **new** HashSet<Integer>();
14. TableDescription desc = configurations.getTableDescription(tableInfo
15. .getOrgName());
16. List<NameNodeHolder> nameNodes = desc.getNameNodes();
17. Function func = desc.getFunction();
18. //分库分表函数，本质就是获取table节点中namenode节点的序号
19. **for** (Map<String, Object> item : parameters) {
20. checkParameters(item, batchItem);
21. **int** i = func.execute(nameNodes.size(), item);
22. indexs.add(i);
23. }
25. **if** (indexs.size() > 0) {
26. AnalyzeResult analyzeResult = batchItem.getAnalyzeResult();
27. HavingInfo havingInfo = analyzeResult.getHavingInfo();
28. **if** (havingInfo != ) {
29. AnalyzerCallback callback = havingInfo.getCallback();
30. **if** (callback != ) {
31. callback.call();
32. }
33. }
34. }
35. **if** ((!batchItem.getAnalyzeResult().getAppendResultColumns().isEmpty() || batchItem
36. .getAnalyzeResult().getLimit() != ) && indexs.size() > 1) {// 存在limit或者avg等聚集函数,需要重新生成sql
37. Collection<AnalyzerCallback> analyzerCallbacks = batchItem
38. .getAnalyzeResult().getAnalyzerCallbacks();
39. **if** (batchItem.getAnalyzeResult().getLimit() != ) {
40. SqlValueItem limitItem = batchItem.getAnalyzeResult()
41. .getLimit();
42. SqlValueItem offsetItem = batchItem.getAnalyzeResult()
43. .getOffset();
44. **if**(offsetItem==){
45. offsetItem=**new** SqlValueItem();
46. offsetItem.setValue(0);
47. }
48. **if** (limitItem.getParameterIndex() > 0
49. && offsetItem.getParameterIndex() > 0) {// limit ?,?
50. Integer limitSize = limitItem.getValue()
51. + offsetItem.getValue();
52. batchItem.getCallback(limitItem.getParameterIndex())
53. .setParameter(limitSize);
54. batchItem.getCallback(offsetItem.getParameterIndex())
55. .setParameter(0);
57. **if**(limitItem.getParameterIndex() > offsetItem.getParameterIndex()){
58. batchItem.getCallback(offsetItem.getParameterIndex())
59. .setParameterIndex(limitItem.getParameterIndex());
61. batchItem.getCallback(limitItem.getParameterIndex())
62. .setParameterIndex(offsetItem.getParameterIndex());
63. }
64. } **else** **if** (limitItem.getParameterIndex() > 0) {// limit 1,?
65. Integer limitSize = limitItem.getValue()
66. + offsetItem.getValue();
67. batchItem.getCallback(limitItem.getParameterIndex())
68. .setParameter(limitSize);
69. } **else** **if** (offsetItem.getParameterIndex() > 0) {// limit ?,10
70. batchItem.getCallback(offsetItem.getParameterIndex())
71. .setParameter(0);
72. }
73. }
74. **for** (AnalyzerCallback callback : analyzerCallbacks) {
75. callback.call();
76. }
77. } **else** {// 在单库路由的情况下，如果检测到是limit ?,?，就置换Parameter顺序
78. SqlValueItem limitItem = batchItem.getAnalyzeResult()
79. .getLimit();
80. SqlValueItem offsetItem = batchItem.getAnalyzeResult()
81. .getOffset();
83. **if**(limitItem != && offsetItem != &&
84. limitItem.getParameterIndex() > offsetItem.getParameterIndex()){
86. batchItem.getCallback(offsetItem.getParameterIndex())
87. .setParameterIndex(limitItem.getParameterIndex());
89. batchItem.getCallback(limitItem.getParameterIndex())
90. .setParameterIndex(offsetItem.getParameterIndex());
91. }
92. }
93. **for** (Integer i : indexs) {// 生成target
94. NameNode nameNode = configurations.getNameNode(
95. tableInfo.getOrgName(), i);
96. DefaultRouteTarget target = **this**.createTarget(batchItem, nameNode, tableInfo);
97. targetSet.add(target);
98. }

101. **for** (RouteTarget item : targetSet) {
102. DefaultRouteTarget target = (DefaultRouteTarget) item;
103. SqlExecuteInfo info = **new** SqlExecuteInfo();
104. info.setCallbacks(**new** LinkedHashSet<ParameterCallback<?>>(batchItem
105. .getCallbacks()));
107. **if** (desc.isDifferentName()) {
108. info.setExecuteSql(configurations.getGenerator().generate(
109. (NameNodeHolder) target.getNameNode(),
110. batchItem.getAnalyzeResult()));
111. } **else** **if** ((!batchItem.getAnalyzeResult().getAppendResultColumns().isEmpty() || batchItem
112. .getAnalyzeResult().getLimit() != ) && nameNodes.size() > 1) {// 存在limit或者avg等聚集函数,需要重新生成sql,必须要超过1个路由结果
113. info.setExecuteSql(configurations.getLimitAvgGenerator()
114. .generate((NameNodeHolder) target.getNameNode(),
115. batchItem.getAnalyzeResult()));
116. } **else** {
117. info.setExecuteSql(batchItem.getSql());
118. }
119. target.setExecuteInfo(info);
120. }
121. **return** targetSet;
122. }

(3)SimpleExecutor 和 HandlerFactory: 根据StatementContext的RouteTarget（路由数据）， 新建事务并获取数据库连接，实际执行JDBC curd操作的类。

**Java代码  [收藏代码](javascript:void())**

1. @SuppressWarnings({ "rawtypes", "unchecked" })
2. **public** **class** SimpleExecutor **implements** Executor {
3. **static** Logger logger = LoggerFactory.getLogger(SimpleExecutor.**class**);
4. **static** **final** ExecuteHandler<Integer> deleteHandler = **new** DeleteExecuteHandler();
5. **static** **final** ExecuteHandler<Integer> insertHandler = **new** InsertExecuteHandler();
6. **static** **final** ExecuteHandler<Integer> updateHandler = **new** UpdateExecuteHandler();
7. **static** **final** ExecuteHandler<ResultSet> queryHandler = **new** QueryExecuteHandler();
9. @Override
10. **public** Object execute(StatementContext context, ExecuteCallback callback)
11. **throws** SQLException {
13. **switch** (context.getCurrentBatch().getAnalyzeResult().getStatementType()) {
14. **case** SELECT:
15. **return** **this**.doQuery(context, callback);
16. **case** INSERT:
17. **case** UPDATE:
18. **case** DELETE:
19. **return** doUpdate(context, callback);
20. **default**:
21. **break**;
22. }
23. **return** ;
24. }
26. ExecuteHandler<?> getHandler(StatementType statementType) {
27. **switch** (statementType) {
28. **case** SELECT:
29. **return** queryHandler;
30. **case** INSERT:
31. **return** insertHandler;
32. **case** UPDATE:
33. **return** updateHandler;
34. **case** DELETE:
35. **return** deleteHandler;
36. **default**:
37. **break**;
38. }
39. **return** ;
40. }
41. ...