SemanticTranslator

`DefaultSemanticTranslator` 类是一个用于将查询语句从业务层转换为实际执行的 SQL 语句的翻译器。它包含以下主要功能：

1. **解析查询**：将业务层的查询参数转换为 SQL 查询。
2. **优化查询**：对生成的 SQL 查询进行优化。
3. **执行转换**：调用具体的转换器将查询参数转换为 SQL 语句。

主要方法及其功能

`translate`

这是整个翻译过程的入口方法，包含解析和优化两个步骤。

```
public void translate(QueryStatement queryStatement) {
    try {
        parse(queryStatement);
        optimize(queryStatement);
        queryStatement.setOk(true);
    } catch (Exception e) {
        queryStatement.setOk(false);
    }
}
```

- **输入参数**：`queryStatement`，包含业务层的查询参数。
- **步骤**：
    1. 调用 `parse` 方法解析查询参数。
    2. 调用 `optimize` 方法优化生成的 SQL 查询。
    3. 设置 `queryStatement` 的状态为成功或失败。

`optimize`

调用所有的查询优化器对 SQL 查询进行优化。

QueryOptimizer

QueryOptimizer

```
public void optimize(QueryStatement queryStatement) {
    for (QueryOptimizer queryOptimizer : ComponentFactory.g
etQueryOptimizers()) {
        queryOptimizer.rewrite(queryStatement);
    }
}
```

- **输入参数**：`queryStatement`，包含生成的 SQL 查询。
- **步骤**：
  1. 获取所有的查询优化器。
  2. 对每个优化器调用 `rewrite` 方法优化 SQL 查询。

`parse`

解析查询参数，将其转换为 SQL 查询。

```
public void parse(QueryStatement queryStatement) throws Exc
eption {
    QueryParam queryParam = queryStatement.getQueryParam();
    if (Objects.isNull(queryStatement.getDataSetQueryParam
())) {
        queryStatement.setDataSetQueryParam(new DataSetQuer
yParam());
    }
    if (Objects.isNull(queryStatement.getMetricQueryParam
())) {
        queryStatement.setMetricQueryParam(new MetricQueryP
aram());
    }
    log.debug("SemanticConverter before [{}]", queryParam);
    for (QueryConverter headlessConverter : ComponentFactor
y.getQueryConverters()) {
```

```
        if (headlessConverter.accept(queryStatement)) {
            log.debug("SemanticConverter accept [{}]", head
lessConverter.getClass().getName());
            headlessConverter.convert(queryStatement);
        }
    }
    log.debug("SemanticConverter after {} {} {}", queryPara
m, queryStatement.getDataSetQueryParam(),
            queryStatement.getMetricQueryParam());
    if (!queryStatement.getDataSetQueryParam().getSql().isE
mpty()) {
        doParse(queryStatement.getDataSetQueryParam(), quer
yStatement);
    } else {
        queryStatement.getMetricQueryParam().setNativeQuery
(queryParam.getQueryType().isNativeAggQuery());
        doParse(queryStatement);
    }
    if (StringUtils.isEmpty(queryStatement.getSql())
            || StringUtils.isEmpty(queryStatement.getSource
Id())) {
        throw new RuntimeException("parse Exception: " + qu
eryStatement.getErrMsg());
    }
    if (StringUtils.isNotBlank(queryStatement.getSql())
            && !SqlSelectHelper.hasLimit(queryStatement.get
Sql())) {
        String querySql = queryStatement.getSql() + " limit
" + queryStatement.getLimit().toString();
        queryStatement.setSql(querySql);
    }
}
```

- **输入参数**：`queryStatement`，包含业务层的查询参数。
- **步骤**：
  1. 初始化 `DataSetQueryParam` 和 `MetricQueryParam`。
  2. 调用所有的查询转换器对 `queryStatement` 进行转换。

3. 根据转换后的查询参数生成 SQL 语句。
4. 检查生成的 SQL 语句是否包含限制条件，如果没有则添加默认的限制条件。

`doParse`

这是 `parse` 方法中的核心逻辑，根据不同的查询参数生成 SQL 语句。

```
public QueryStatement doParse(DataSetQueryParam dataSetQuer
yParam, QueryStatement queryStatement) {
    log.info("parse dataSetQuery [{}] ", dataSetQueryPara
m);
    try {
        if (!CollectionUtils.isEmpty(dataSetQueryParam.getT
ables())) {
            List<String[]> tables = new ArrayList<>();
            boolean isSingleTable = dataSetQueryParam.getTa
bles().size() == 1;
            for (MetricTable metricTable : dataSetQueryPara
m.getTables()) {
                QueryStatement tableSql = parserSql(metricT
able, isSingleTable,
                        dataSetQueryParam, queryStatement);
                if (isSingleTable && Objects.nonNull(tableS
ql.getDataSetQueryParam())
                        && !tableSql.getDataSetSimplifySql
().isEmpty()) {
                    queryStatement.setSql(tableSql.getDataS
etSimplifySql());
                    queryStatement.setDataSetQueryParam(dat
aSetQueryParam);
                    return queryStatement;
                }
                tables.add(new String[]{metricTable.getAlia
s(), tableSql.getSql()});
            }
            if (!tables.isEmpty()) {
                String sql;
                if (dataSetQueryParam.isSupportWith()) {
                    sql = "with " + tables.stream().map(t -
```

```
> String.format("%s as (%s)", t[0], t[1])).collect(
                            Collectors.joining(",")) + "\n"
+ dataSetQueryParam.getSql();
                } else {
                    sql = dataSetQueryParam.getSql();
                    for (String[] tb : tables) {
                        sql = StringUtils.replace(sql, tb
[0],
                                "(" + tb[1] + ") " + (dataS
etQueryParam.isWithAlias() ? "" : tb[0]), -1);
                    }
                }
                queryStatement.setSql(sql);
                queryStatement.setDataSetQueryParam(dataSet
QueryParam);
                return queryStatement;
            }
        }
    } catch (Exception e) {
        log.error("physicalSql error {}", e);
        queryStatement.setErrMsg(e.getMessage());
    }
    return queryStatement;
}
```

- **输入参数**：`dataSetQueryParam` 和 `queryStatement`。
- **步骤**：
  1. 处理 `dataSetQueryParam` 中的所有表，将其转换为 SQL 语句。
  2. 如果只有一个表且解析成功，则直接返回解析后的 SQL 语句。
  3. 如果有多个表，则生成带有 WITH 子句的 SQL 语句。
  4. 将生成的 SQL 语句设置到 `queryStatement` 中。

`parserSql`

这是一个辅助方法，用于解析每个表的 SQL 语句。

```
private QueryStatement parserSql(MetricTable metricTable, B
oolean isSingleMetricTable,
                                 DataSetQueryParam dataSetQ
ueryParam,
                                 QueryStatement queryStatem
ent) throws Exception {
    MetricQueryParam metricReq = new MetricQueryParam();
    metricReq.setMetrics(metricTable.getMetrics());
    metricReq.setDimensions(metricTable.getDimensions());
    metricReq.setWhere(StringUtil.formatSqlQuota(metricTabl
e.getWhere()));
    metricReq.setNativeQuery(!AggOption.isAgg(metricTable.g
etAggOption()));
    QueryStatement tableSql = new QueryStatement();
    tableSql.setIsS2SQL(false);
    tableSql.setMetricQueryParam(metricReq);
    tableSql.setMinMaxTime(queryStatement.getMinMaxTime());
    tableSql.setEnableOptimize(queryStatement.getEnableOpti
mize());
    tableSql.setDataSetId(queryStatement.getDataSetId());
    tableSql.setSemanticModel(queryStatement.getSemanticMod
el());
    if (isSingleMetricTable) {
        tableSql.setDataSetSql(dataSetQueryParam.getSql());
        tableSql.setDataSetAlias(metricTable.getAlias());
    }
    tableSql = doParse(tableSql, metricTable.getAggOption
());
    if (!tableSql.isOk()) {
        throw new Exception(String.format("parser table [%
s] error [%s]", metricTable.getAlias(),
                tableSql.getErrMsg()));
    }
    queryStatement.setSourceId(tableSql.getSourceId());
    return tableSql;
}
```

- **输入参数**：`metricTable`，`isSingleMetricTable`，`dataSetQueryParam`，`queryStatement`。
- **步骤**：
  1. 初始化 `MetricQueryParam`，设置度量、维度和过滤条件。
  2. 创建新的 `QueryStatement`，设置必要的属性。
  3. 调用 `doParse` 方法解析 SQL 语句。
  4. 如果解析失败，抛出异常。
  5. 设置 `queryStatement` 的数据源 ID。

总结

`DefaultSemanticTranslator` 类的主要功能是将业务层的查询参数转换为实际执行的 SQL 语句，并对生成的 SQL 语句进行优化。通过调用具体的查询转换器和优化器，实现查询语句的解析和优化。以下是一个简化的示例：

示例

假设我们有一个查询参数，包含一个度量和一个维度：

```
QueryParam queryParam = new QueryParam();
queryParam.setMetrics(Arrays.asList("metric1"));
queryParam.setDimensions(Arrays.asList("dimension1"));

QueryStatement queryStatement = new QueryStatement();
queryStatement.setQueryParam(queryParam);
```

调用 `translate` 方法将其转换为 SQL 语句：

```
DefaultSemanticTranslator translator = new DefaultSemanticT
ranslator();
translator.translate(queryStatement);

System.out.println("Generated SQL: " + queryStatement.getSq
l());
```

输出示例：

```
Generated SQL: SELECT dimension1, SUM(metric1) FROM my_tabl
e GROUP BY dimension1
```

通过理解 `DefaultSemanticTranslator` 类的实现，可以深入了解查询参数是如何被转换为 SQL 语句并进行优化的。