`LMSqlParser` + `QueryTypeParser`

`LLMSqlParser` 类主要用于通过大型语言模型（LLM）理解查询语义并生成 S2SQL 语句，这些语句由语义查询引擎执行。下面是对该类的详细解析：

主要职责

`LLMSqlParser` 类的主要职责是：

1. 通过调用大型语言模型服务，解析用户输入的自然语言查询。
2. 基于解析结果生成 S2SQL 语句。
3. 对生成的多个 S2SQL 语句去重，并根据权重选择最优的语句用于执行。

关键方法解析

`parse(QueryContext queryCtx, ChatContext chatCtx)`

这是实现 `SemanticParser` 接口的主要方法，该方法执行如下操作：

1. **获取 `LLMRequestService` 实例**：通过 `ContextUtils.getBean(LLMRequestService.class)` 获取 `LLMRequestService` 实例，用于后续处理请求。
2. **判断是否跳过解析**：调用 `requestService.isSkip(queryCtx)` 方法确定是否跳过该解析器。
3. **获取数据集 ID**：通过 `requestService.getDataSetId(queryCtx)` 获取当前上下文中的数据集 ID。如果无法获取，则直接返回，不继续解析。
4. **调用 LLM 服务进行解析**：调用 `tryParse(queryCtx, dataSetId)` 方法，实际执行调用 LLM 服务进行解析的逻辑。

`tryParse(QueryContext queryCtx, Long dataSetId)`

这个方法是核心逻辑所在：

1. **获取相关服务实例**：获取 `LLMRequestService`、`LLMResponseService` 以及 `LLMParserConfig` 实例，分别用于请求、响应处理和配置管理。
2. **构建 LLM 请求**：使用 `requestService.getLlmReq(queryCtx, dataSetId)` 构建 LLM 请求对象 `LLMReq`。
3. **循环重试**：基于配置的最大重试次数，循环尝试调用 LLM 服务：
   - **调用 LLM 服务**：使用 `requestService.runText2SQL(llmReq)` 发送请求并获取 LLM 响应 `LLMResp`。

- **去重并解析**：调用 `responseService.getDeduplicationSqlResp(currentRetry, llmResp)` 方法对返回的 SQL 结果进行去重，并构建 `ParseResult` 对象。
- **检查并记录解析结果**：如果有可用的 SQL 结果，则记录解析结果并结束循环。

4. **处理结果**：如果存在有效的 SQL 结果，则通过 `responseService.addParseInfo` 方法将解析信息添加到 `QueryContext` 中。

类的意义

`LLMSqlParser` 通过与 LLM 的交互，使得系统能够理解复杂的自然语言查询，并将其转换为结构化的 SQL 语句。这一过程充分利用了 LLM 的强大能力，并且通过多次重试和去重机制，确保生成的 SQL 语句质量较高，能够满足用户的查询需求。

结合 `QueryContext` 和 `ChatContext`

- `QueryContext`: 主要用于保存查询的上下文信息，包括查询文本、用户信息、数据集 ID 等。这些信息被传递给 LLM，用于生成 SQL 语句。
- `ChatContext`: 用于存储与当前对话相关的上下文信息，比如查询文本和解析信息。

在 `LLMSqlParser` 中，`QueryContext` 和 `ChatContext` 被用来维持与用户查询的语义信息，从而生成合适的 SQL 语句。这两个上下文对象携带了丰富的元数据，使得整个解析过程更加准确和高效。

总结

`LLMSqlParser` 通过与 LLM 服务的交互，将自然语言转换为结构化查询语言（SQL），以满足用户的查询需求。它通过多次重试、去重和选择最优 SQL 语句的方式，保证了生成结果的准确性和可靠性。

`QueryTypeParser`

`QueryContext` 类解析

`QueryContext` 类主要负责保存与查询相关的上下文信息，它包含了一些关键字段，如 `queryText`（用户输入的查询文本）、`dataSetIds`（数据集的 ID 集合）、`candidateQueries`（候选查询列表）、`semanticSchema`（语义模式）等。它的主要作用是为查询解析和执行提供必要的上下文信息。

关键字段说明：

- **queryText**：用户输入的自然语言查询文本。
- **dataSetIds**：与查询相关的数据集的 ID。

- **modelIdToDataSetIds**：模型 ID 与数据集 ID 的映射关系。
- **user**：当前执行查询的用户信息。
- **text2SQLType**：表示将文本转化为 SQL 的方式（如通过规则或 LLM）。
- **candidateQueries**：候选的语义查询列表，通过解析后生成的可能查询。
- **semanticSchema**：语义模式，用于查询解析时的语义理解。
- **workflowState**：当前查询的工作流状态。
- **llmConfig**：与大语言模型（LLM）相关的配置。
- **exemplars**：示例 SQL，用于支持或参考生成 SQL 的过程。

`getCandidateQueries` 方法根据某种排序标准（通常是解析评分）对候选查询进行排序，并限制返回的数量。这说明在 `QueryTypeParser` 执行前，`QueryContext` 已经包含多个经过不同程度解析的候选查询。

`ChatContext` 类解析

`ChatContext` 类则用于保存聊天上下文信息，通常是与用户交互相关的内容。它包括查询的 ID、查询文本以及 `SemanticParseInfo`（语义解析信息）。

关键字段说明：

- **chatId**：聊天的 ID，用于唯一标识一个会话。
- **queryText**：用户的查询文本。
- **parseInfo**：包含了语义解析的详细信息（如生成的 SQL、解析评分等）。
- **user**：执行查询的用户。

这个类的主要目的是跟踪与特定聊天会话相关的查询解析状态。

`QueryTypeParser` 类

`QueryTypeParser` 类的主要功能是通过对 `QueryContext` 中的候选查询进行进一步解析和分类，确定每个查询的类型（如 METRIC、TAG、ID 或 DETAIL）。

在上下文中：

- **初始化 S2SQL**：在 `parse` 方法中，首先通过调用 `semanticQuery.initS2Sql()` 初始化每个候选查询的 SQL。这一步依赖于 `QueryContext` 中的 `semanticSchema` 和 `user` 信息。`initS2Sql` 会使用这些上下文信息生成初步的 SQL 查询。
- **设置查询类型**：通过调用 `getQueryType()` 方法，`QueryTypeParser` 对 SQL 查询进行分类。`getQueryType()` 方法会根据 `SqlInfo` 中的信息来判断查询类型。这个过程中使用了 `SemanticSchema` 来检查 `select` 和 `where` 子句中的字段是否与度量（metric）、实体（entity）、标签（tag）或时间维度（time dimension）相关。

结合上下文，我们可以看到：

- `QueryContext` 提供了丰富的上下文信息，如候选查询、语义模式、用户信息等，供 `QueryTypeParser` 在解析过程中使用。
- `ChatContext` 提供了与当前查询相关的会话信息，并记录了初步解析的结果，便于 `QueryTypeParser` 进一步分类。

`QueryTypeParser` 类中的 `parse` 方法

```
@Override
public void parse(QueryContext queryContext, ChatContext chatC
ontext) {
    List<SemanticQuery> candidateQueries = queryContext.getCan
didateQueries();
    User user = queryContext.getUser();

    for (SemanticQuery semanticQuery : candidateQueries) {
        // 1. init S2SQL
        semanticQuery.initS2Sql(queryContext.getSemanticSchema
(), user);
        // 2. set queryType
        QueryType queryType = getQueryType(queryContext, seman
ticQuery);
        semanticQuery.getParseInfo().setQueryType(queryType);
    }
}
```

- **数据流**：`QueryContext` 中的 `candidateQueries` 被逐一处理，每个查询都被初始化S2SQL，随后通过 `getQueryType()` 进行类型分类。
- **候选查询的处理**：在 `QueryContext` 中，候选查询是通过某种排序机制生成的，`QueryTypeParser` 进一步对这些查询进行分类，确定其类型。
- **SQL校正和优化**：
  - **时间戳**：14:57:19
  - 在接下来的步骤中，通过多个语义校正器（如 `WhereCorrector`，`GroupByCorrector`，`AggCorrector` 等），对初步生成的SQL语句进行校正和优化。
  - `supersonic\headless\chat\src\main\java\com\tencent\supersonic\headless\chat\corrector`
  - 这些校正步骤包括确保SQL语句在语法和逻辑上的正确性，例如添加WHERE子句以确保日期范围正确，调整聚合函数和分组条件等，以确保查询结果的准确性和

相关性。

- **最终SQL输出**：
  - 输出修正后的SQL语句，确保它完全符合业务需求和数据查询的正确性。

流程分析

`LLMSqlParser` 的职责

`LLMSqlParser` 是一个用于处理和解析自然语言查询的组件，它利用大型语言模型（LLM）将自然语言查询转换为 SQL 查询。具体来说，`LLMSqlParser` 负责以下任务：

1. **调用 LLM 服务**：通过调用 LLM 服务来生成 SQL 查询。
2. **多轮重试**：如果第一次解析失败，它会重试多次以确保生成的 SQL 查询是最佳的。
3. **去重**：在多轮重试过程中，它会对生成的 SQL 结果进行去重处理。
4. **生成解析信息**：将最终的解析结果（SQL 查询）存储在 `QueryContext` 中。

`QueryTypeParser` 的职责

`QueryTypeParser` 的主要任务是确定 SQL 查询的类型，即 `QueryType`。根据解析后的 SQL 查询的内容，它会将查询分类为以下类型之一：

1. **ID 查询**：当 SQL 查询的 WHERE 子句包含时间字段，并且这些字段与实体 ID 相关联时，查询类型被归类为 ID 查询。
2. **METRIC 查询**：当 SQL 查询的 SELECT 子句中包含度量（Metric）字段时，查询类型被归类为 METRIC 查询。
3. **DETAIL 查询**：如果 SQL 查询不符合以上条件，则被归类为 DETAIL 查询。

两者的关联与关系

1. **流程中的位置**：在查询处理流程中，`LLMSqlParser` 通常会先于 `QueryTypeParser` 运行。`LLMSqlParser` 生成 SQL 查询后，这些查询会被传递给 `QueryTypeParser`，后者对生成的 SQL 查询进行类型判断。
2. **依赖关系**：`QueryTypeParser` 依赖于 `LLMSqlParser` 生成的 SQL 查询进行分析。因此，`QueryTypeParser` 的输入实际上是 `LLMSqlParser` 生成的输出。
3. **查询优化与理解**：`LLMSqlParser` 主要关注的是将自然语言转换为 SQL 查询，而 `QueryTypeParser` 则进一步理解和分类这些 SQL 查询，以便后续的查询执行和优化。这种分类可以影响查询的执行计划，进而影响查询性能和结果的准确性。