服务器出问题，目前部分恢复

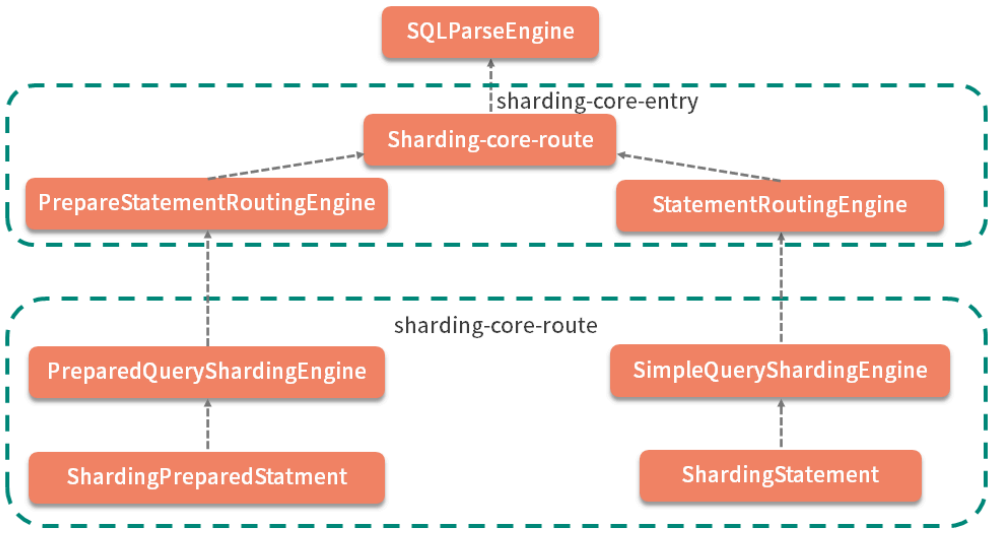
17 路由引擎：如何理解分片路由核心类 ShardingRouter 的运作机制?

前面我们花了几个课时对 ShardingSphere 中的 SQL 解析引擎做了介绍，我们明白 SQL 解析的作用就是根据输入的 SQL 语句生成一个 SQLStatement 对象。

从今天开始，我们将进入 **ShardingSphere 的路由（Routing）引擎部分的源码解析**。从流程上讲，**路由引擎**是整个分片引擎执行流程中的第二步，即基于 SQL 解析引擎所生成的 SQLStatement，通过解析执行过程中所携带的上下文信息，来获取匹配数据库和表的分片策略，并生成路由结果。

分层：路由引擎整体架构

与介绍 SQL 解析引擎时一样，我们通过翻阅 ShardingSphere 源码，首先梳理了如下所示的包结构：



路由引擎核心类的分层分包图

上述包图总结了与路由机制相关的各个核心类，我们可以看到整体呈一种对称结构，即根据是 **PreparedStatement** 还是**普通 Statement** 分成两个分支流程。

同时，我们也可以把这张图中的类按照其所属的包结构**分成两个层次**：位于底层的 sharding-core-route 和位于上层的 sharding-core-entry，这也是 ShardingSphere 中所普遍采用的一

种分包原则，即**根据类的所属层级来组织包结构**。关于 ShardingSphere 的分包原则我们在[《12 | 从应用到原理：如何高效阅读 ShardingSphere 源码？》]中也已经进行了介绍，接下来我们具体分析这一原则在路由引擎中的应用。

1.sharding-core-route 工程

我们先来看图中的 ShardingRouter 类，该类是整个路由流程的启动点。ShardingRouter 类直接依赖于解析引擎 SQLParseEngine 类完成 SQL 解析并获取 SQLStatement 对象，然后供 PreparedStatementRoutingEngine 和 StatementRoutingEngine 进行使用。注意到这几个类都位于 sharding-core-route 工程中，**处于底层组件**。

2.sharding-core-entry 工程

另一方面，上图中的 PreparedQueryShardingEngine 和 SimpleQueryShardingEngine 则位于 sharding-core-entry 工程中。从包的命名上看，entry 相当于是访问的入口，所以我们可以判断这个工程中所提供的类**属于面向应用层组件**，处于更加上层的位置。PreparedQueryShardingEngine 和 SimpleQueryShardingEngine 的使用者分别是 ShardingPreparedStatement 和 ShardingStatement。这两个类再往上就是 ShardingConnection 以及 ShardingDataSource 这些直接面向应用层的类了。

路由核心类：ShardingRouter

通过以上分析，我们对路由引擎的整体结构有了一个初步的认识。对于采用分层结构的执行流程而言，有两种解析思路，即自上而下或自下而上。今天，我们的思路是**从底层出发逐层往上**分析流程的链路，先来看路由引擎中最底层的对象 ShardingRouter，变量定义如下：

```
private final ShardingRule shardingRule;
private final ShardingSphereMetaData metaData;
private final SQLParseEngine parseEngine;
```

在 ShardingRouter 中，我们首先看到了熟悉的 SQL 解析引擎 SQLParseEngine 以及它的使用方法：

```
public SQLStatement parse(final String logicSQL, final boolean useCache) {
        return parseEngine.parse(logicSQL, useCache);
}
```

上述代码非常简单，即通过 SQLParseEngine 对传入的 SQL 进行解析返回一个 SQLStatement 对象。这里将 SQL 命名为 logicSQL，以便区别在分片和读写分离情况下的真实 SQL。

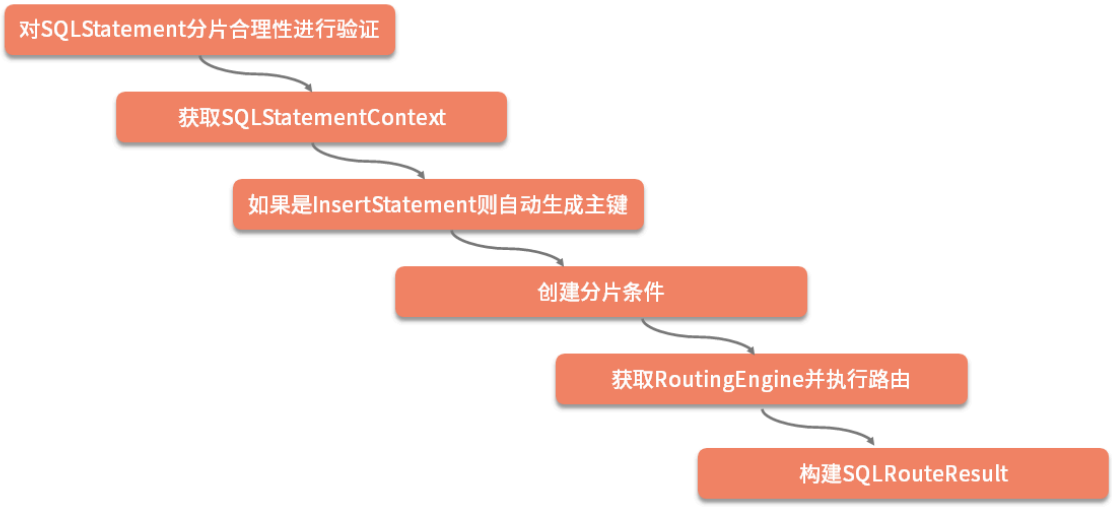
接下来我们来看一下 ShardingRule，请注意这是一个基础类，代表着分片的各种规则信息。ShardingRule 类位于 sharding-core-common 工程中，主要保存着与分片相关的各种规则信

息，以及 ShardingKeyGenerator 等分布式主键的创建过程，各个变量定义以及对应的注释如下所示：

```
//分片规则配置类，封装各种配置项信息
private final ShardingRuleConfiguration ruleConfiguration;
//DataSource 名称列表
private final ShardingDataSourceNames shardingDataSourceNames;
//针对表的规则列表
private final Collection<TableRule> tableRules;
//针对绑定表的规则列表
private final Collection<BindingTableRule> bindingTableRules;
//广播表名称列表
private final Collection<String> broadcastTables;
//默认的数据库分片策略
private final ShardingStrategy defaultDatabaseShardingStrategy;
//默认的数据表分片策略
private final ShardingStrategy defaultTableShardingStrategy;
//默认的分片键生成器
private final ShardingKeyGenerator defaultShardingKeyGenerator;
//针对读写分离的规则列表
private final Collection<MasterSlaveRule> masterSlaveRules;
//加密规则
private final EncryptRule encryptRule;
```

ShardingRule 的内容非常丰富，但其定位更多是提供规则信息，而不属于核心流程，因此我们先不对其做详细展开。作为基础规则类，ShardingRule 会贯穿整个分片流程，在后续讲解过程中我们会穿插对它的介绍，这里先对上述变量的名称和含义有简单认识即可。

我们回到 ShardingRouter 类，发现其核心方法只有一个，即 route 方法。这个方法的逻辑比较复杂，我们梳理它的执行步骤，如下图所示：



ShardingRouter 的 route 方法核心步骤

ShardingRouter 是路由引擎的核心类，**在接下来的内容中，我们将对上图中的 6 个步骤分别一一详细展开，帮忙你理解一个路由引擎的设计思想和实现机制。**

1.分片合理性验证

我们首先来看 ShardingRouter 的第一个步骤，即验证分片信息的合理性，验证方式如下所示：

```
//使用ShardingStatementValidator对Statement进行验证
Optional<ShardingStatementValidator> shardingStatementValidator = ShardingSta
if (shardingStatementValidator.isPresent()) {
     shardingStatementValidator.get().validate(shardingRule, sqlStatement, pa
}
```

这段代码使用 ShardingStatementValidator 对输入的 SQLStatement 进行验证，可以看到这里用到了典型的工厂模式，工厂类 ShardingStatementValidatorFactory 如下所示：

```
public final class ShardingStatementValidatorFactory {

    public static Optional<ShardingStatementValidator> newInstance(final SQLS
        if (sqlStatement instanceof InsertStatement) {
            return Optional.<ShardingStatementValidator>of(new ShardingInsert
        }
        if (sqlStatement instanceof UpdateStatement) {
            return Optional.<ShardingStatementValidator>of(new ShardingUpdate
        }
        return Optional.absent();
    }
}
```

注意到 ShardingStatementValidator 要验证的只有 InsertStatement 和 UpdateStatement 这两个 SQLStatement。那么如何进行验证呢？我们来看一下 ShardingStatementValidator 的定义，如下所示：

```
public interface ShardingStatementValidator<T extends SQLStatement> {

    //验证分片操作是否支持
    void validate(ShardingRule shardingRule, T sqlStatement, List<Object> par
}
```

对于验证过程而言，核心思想在于根据 SQLStatement 中的 Segment 与 ShardingRule 中的规则来判断它们之间是否有需要特殊处理的判断逻辑。我们以 ShardingInsertStatementValidator 为例来看验证过程，它的 validate 方法如下所示：

```
public final class ShardingInsertStatementValidator implements ShardingStatem

    @Override
    public void validate(final ShardingRule shardingRule, final InsertStateme
        Optional<OnDuplicateKeyColumnsSegment> onDuplicateKeyColumnsSegment =
```

```
            //如果是"ON DUPLICATE KEY UPDATE"语句，且如果当前操作的是分片Column时，验证不
            if (onDuplicateKeyColumnsSegment.isPresent() && isUpdateShardingKey(s
                throw new ShardingException("INSERT INTO .... ON DUPLICATE KEY UP
            }
        }
        …
    }
```

可以看到这里的判断逻辑与“ON DUPLICATE KEY UPDATE”这一 Mysql 特有的语法相关，该语法允许我们通过 Update 的方式插入有重复主键的数据行（实际上这个语法也不是常规语法，本身也不大应该被使用）。

ShardingInsertStatementValidator 先判断是否存在 OnDuplicateKeyColumn，然后再判断这个 Column 是否是分片键，如果同时满足这两个条件，则直接抛出一个异常，不允许在分片 Column 上执行“INSERT INTO …. ON DUPLICATE KEY UPDATE”语法。

2.获取上下文

接下来我们来看 ShardingRouter 类中 route 方法的第二段代码，该段代码比较简单，用于获取运行时的 SQLStatement 上下文，如下所示：

```
//获取 SQLStatementContext
SQLStatementContext sqlStatementContext = SQLStatementContextFactory.newInsta
```

可以看到这里构建了上下文对象 SQLStatementContext，同样用到了工厂模式，工厂类 SQLStatementContextFactory 如下所示：

```
public final class SQLStatementContextFactory {

    public static SQLStatementContext newInstance(final RelationMetas relatio
        if (sqlStatement instanceof SelectStatement) {
            return new SelectSQLStatementContext(relationMetas, sql, paramete
        }
        if (sqlStatement instanceof InsertStatement) {
            return new InsertSQLStatementContext(relationMetas, parameters, (
        }
        return new CommonSQLStatementContext(sqlStatement);
    }
}
```

请注意 SQLStatementContext 只有三种：

- SelectSQLStatementContext
- InsertSQLStatementContext
- CommonSQLStatementContext

它们都实现了 SQLStatementContext 接口，顾名思义，所谓的 **SQLStatementContext 就是一种上下文对象**，保存着与特定 SQLStatement 相关的上下文信息，用于为后续处理提供数据存储和传递的手段。

我们可以想象在 SQLStatementContext 中势必都持有 SQLStatement 对象以及与表结构信息相关的上下文 TablesContext。

对于 SelectSQLStatement，通常也需要保存与查询相关的分组上下文 GroupByContext、排序上下文 OrderByContext 和分页上下文 PaginationContext；而对于 InsertSQLStatementContext 而言，InsertValueContext 则包含了所有与插入操作相关的值对象。

3.自动生成主键

接下来的第三段代码与数据库主键相关，同样只有一句代码，如下所示：

```
//如果是 InsertStatement 则自动生成主键
Optional<GeneratedKey> generatedKey = sqlStatement instanceof InsertStatement
            ? GeneratedKey.getGenerateKey(shardingRule, metaData.getTables(),
```

这段代码的逻辑比较明确，即如果输入的 SQLStatement 是 InsertStatement，则自动创建一个主键 GeneratedKey，反之就不做处理。

在数据分片的场景下，创建一个分布式主键实际上并没有那么简单，所以在这段代码背后有很多设计的思想和实现的技巧值得我们进行深入分析，关于这个主题，我们已经在 [《14 | 分布式主键：ShardingSphere 中有哪些分布式主键实现方式？》]中对分布式主键生成机制做了专题分享。

4.创建分片条件

我们来看 ShardingRouter 中 route 方法的第四个步骤，这个步骤的作用是创建分片条件，如下所示：

```
//创建分片条件
ShardingConditions shardingConditions = getShardingConditions(parameters, sql
boolean needMergeShardingValues = isNeedMergeShardingValues(sqlStatementConte
if (sqlStatementContext.getSqlStatement() instanceof DMLStatement && needMerg
    checkSubqueryShardingValues(sqlStatementContext, shardingConditions);
    mergeShardingConditions(shardingConditions);
}
```

在 ShardingSphere 中，分片条件对象 ShardingCondition 定义如下所示，包含了一组路由信息和节点信息，其中路由信息包含表名和列名，而节点信息包含数据源名和表名：

```
public class ShardingCondition {
    //路由信息
    private final List<RouteValue> routeValues = new LinkedList<>();
    //节点信息
    private final Collection<DataNode> dataNodes = new LinkedList<>();
}
```

那么如何获取分片条件呢？如下所示的 getShardingConditions 方法给出了具体的实现方式，可以看到这里根据输入的 SQL 类型，分别通过 InsertClauseShardingConditionEngine 和WhereClauseShardingConditionEngine 创建了 ShardingConditions：

```
private ShardingConditions getShardingConditions(final List<Object> parameter
    if (sqlStatementContext.getSqlStatement() instanceof DMLStatement) {
     //如果是 InsertSQLStatement 上下文
        if (sqlStatementContext instanceof InsertSQLStatementContext) {
            InsertSQLStatementContext shardingInsertStatement = (InsertSQLSta
            //通过 InsertClauseShardingConditionEngine 创建分片条件
            return new ShardingConditions(new InsertClauseShardingConditionEn
        }
        //否则直接通过 WhereClauseShardingConditionEngine 创建分片条件
        return new ShardingConditions(new WhereClauseShardingConditionEngine(
    }
    return new ShardingConditions(Collections.<ShardingCondition>emptyList())
}
```

对于路由引擎而言，分片条件的主要目的就是提取用于路由的目标数据库、表和列之间的关系，InsertClauseShardingConditionEngine 和 WhereClauseShardingConditionEngine 中的处理逻辑都是为了构建包含这些关系信息的一组 ShardingCondition 对象。

当获取这些 ShardingCondition 之后，我们还看到有一个优化的步骤，即调用 mergeShardingConditions，对可以合并的 ShardingCondition 进行合并。

5.执行路由

当我们获取了 SQLStatement 上下文，并创建了分片条件，接下来就是真正执行路由，如下所示：

```
//获取 RoutingEngine 并执行路由
RoutingEngine routingEngine = RoutingEngineFactory.newInstance(shardingRule,
RoutingResult routingResult = routingEngine.route();
```

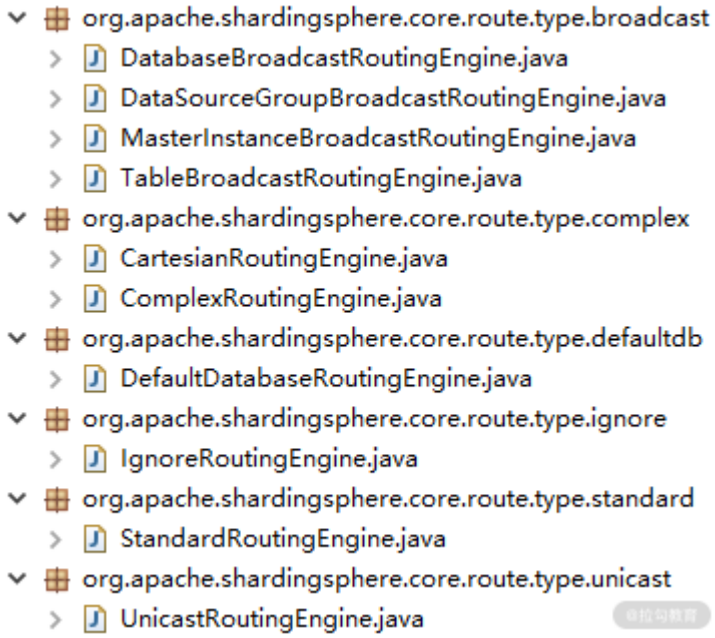
这两句代码是 ShardingRouter 类的核心，我们获取了一个 RoutingEngine 实例，然后基于该实例执行路由并返回一个 RoutingResult 对象。RoutingEngine 定义如下，只有一个简单的 route 方法：

```
public interface RoutingEngine {
    //执行路由
    RoutingResult route();
}
```

在 ShardingSphere 中存在一批 RoutingEngine 的实现类，RoutingEngineFactory 工厂类负责生成这些具体的 RoutingEngine，生成逻辑如下所示：

```
public static RoutingEngine newInstance(final ShardingRule shardingRule,
                                        final ShardingSphereMetaData meta
        SQLStatement sqlStatement = sqlStatementContext.getSqlStatement();
        Collection<String> tableNames = sqlStatementContext.getTablesContext(

        //全库路由
        if (sqlStatement instanceof TCLStatement) {
            return new DatabaseBroadcastRoutingEngine(shardingRule);
        }
        //全库表路由
        if (sqlStatement instanceof DDLStatement) {
            return new TableBroadcastRoutingEngine(shardingRule, metaData.get
        }
        //阻断路由
        if (sqlStatement instanceof DALStatement) {
            return getDALRoutingEngine(shardingRule, sqlStatement, tableNames
        }
        //全实例路由
        if (sqlStatement instanceof DCLStatement) {
            return getDCLRoutingEngine(shardingRule, sqlStatementContext, met
        }
        //默认库路由
        if (shardingRule.isAllInDefaultDataSource(tableNames)) {
            return new DefaultDatabaseRoutingEngine(shardingRule, tableNames)
        }
        //全库路由
        if (shardingRule.isAllBroadcastTables(tableNames)) {
            return sqlStatement instanceof SelectStatement ? new UnicastRouti
        }
        //默认库路由
        if (sqlStatementContext.getSqlStatement() instanceof DMLStatement &&
            return new DefaultDatabaseRoutingEngine(shardingRule, tableNames)
        }
        //单播路由
        if (sqlStatementContext.getSqlStatement() instanceof DMLStatement &&
            return new UnicastRoutingEngine(shardingRule, tableNames);
        }
        //分片路由
        return getShardingRoutingEngine(shardingRule, sqlStatementContext, sh
}
```

这些 RoutingEngine 的具体介绍我们放在下一课时《18 | 路由引擎：如何实现数据访问的分片路由和广播路由?》中进行详细介绍，这里只需要了解 ShardingSphere 在包结构的设计上把具体的 RoutingEngine 分成了六大类：即广播（broadcast）路由、混合（complex）路由、默认数据库（defaultdb）路由、无效（ignore）路由、标准（standard）路由以及单播（unicast）路由，如下所示：



不同类型的 RoutingEngine 实现类

RoutingEngine 的执行结果是 RoutingResult，而 RoutingResult 中包含了一个 RoutingUnit 集合，RoutingUnit 中的变量定义如下所示，可以看到有两个关于 DataSource 名称的变量以及一个 TableUnit 列表：

```
//真实数据源名
private final String dataSourceName;
//逻辑数据源名
private final String masterSlaveLogicDataSourceName;
//表单元列表
private final List<TableUnit> tableUnits = new LinkedList<>();
```

而 TableUnit 保存着逻辑表名和实际表名，如下所示：

```
public final class TableUnit {
    //逻辑表名
    private final String logicTableName;
    //真实表名
    private final String actualTableName;
}
```

所以 RoutingResult 中保存的实际上就是一组关于数据库与数据表的对应关系，其中库与表都存在逻辑值和真实值。

6.构建路由结果

当通过一系列的路由引擎处理之后，我们获得了 RoutingResult 对象，但并不是直接将其进行返回，而是会构建一个 SQLRouteResult 对象。这就是 ShardingRouter 的 route 方法最后一个步骤，如下所示：

```
//构建 SQLRouteResult
SQLRouteResult result = new SQLRouteResult(sqlStatementContext, shardingCondi
result.setRoutingResult(routingResult);
//如果是Insert语句，则设置自动生成的分片键
if (sqlStatementContext instanceof InsertSQLStatementContext) {
    setGeneratedValues(result);
}
return result;
```

我们来到 SQLRouteResult 的定义，看看它与 RouteResult 之间有什么不同，SQLRouteResult中 的变量如下所示：

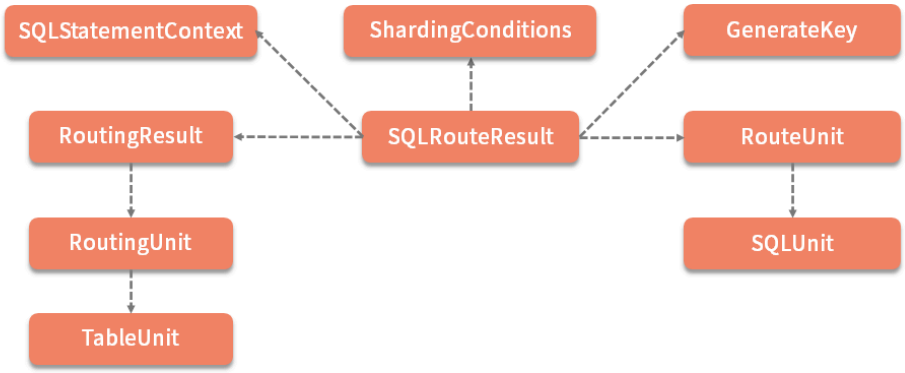
```
//SQLStatement 上下文
private final SQLStatementContext sqlStatementContext;
//分片条件
private final ShardingConditions shardingConditions;
//自动生成的分片键
private final GeneratedKey generatedKey;
//一组路由单元
private final Collection<RouteUnit> routeUnits = new LinkedHashSet<>();
//由 RoutingEngine 生成的 RoutingResult
private RoutingResult routingResult;
```

可以看到 SQLRouteResult 中包含了 RoutingResult。我们可以认为 SQLRouteResult 是整个 SQL 路由返回的路由结果，在后续的流程中还会被 PreparedStatementRoutingEngine 等上层对象所使用，而 RoutingResult 只是 RoutingEngine 返回的路由结果，它的使用者就是位于底层的 ShardingRouter。

同时，我们注意到这里有一个新的 Unit 对象 RouteUnit，包含了数据源名称以及 SQL 单元对象 SQLUnit，如下所示：

```
public final class RouteUnit {
    //数据源名
    private final String dataSourceName;
    //SQL 单元
    private final SQLUnit sqlUnit;
}
```

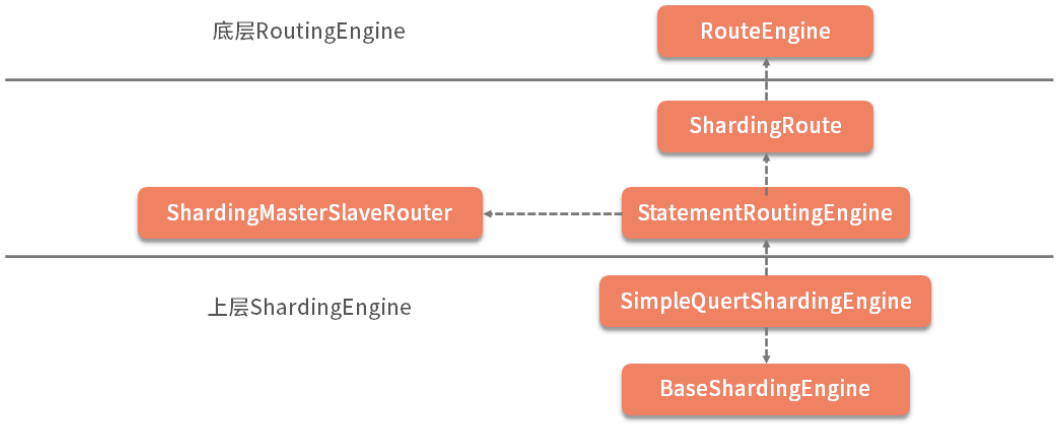
这里的 SQLUnit 中就是最终的一条 SQL 语句以及相应参数的组合。因为路由结果对象 SQLRouteResult 会继续传递到分片引擎的后续流程，且内部结构比较复杂，所以这里通过如下所示的类图对其包含的各种变量进行总结，方便你进行理解。



SQLRouteResult 类层结构

至此，我们把 ShardingRouter 类的核心流程做了介绍。在 ShardingSphere 的路由引擎中，ShardingRouter 可以说是一个承上启下的核心类，向下我们可以挖掘各种 RoutingEngine 的具体实现；向上我们可以延展到读写分离等面向应用的具体场景。

下图展示了 ShardingRouter 的这种定位关系。关于各种 RoutingEngine 的介绍是我们下一课时的内容，今天我们先将基于 ShardingRouter 讨论它的上层结构，从而引出了 ShardingEngine。



ShardingRouter 类的承上启下作用示意图

从底层 ShardingRouter 到上层 ShardingEngine

我们的思路仍然是从下往上，先来看上图中的 StatementRoutingEngine，其实现如下所示：

```
public final class StatementRoutingEngine {

    private final ShardingRouter shardingRouter;

    private final ShardingMasterSlaveRouter masterSlaveRouter;

    public StatementRoutingEngine(final ShardingRule shardingRule, final Shar
        shardingRouter = new ShardingRouter(shardingRule, metaData, sqlParseE
        masterSlaveRouter = new ShardingMasterSlaveRouter(shardingRule.getMas
    }

    public SQLRouteResult route(final String logicSQL) {
        SQLStatement sqlStatement = shardingRouter.parse(logicSQL, false);
        return masterSlaveRouter.route(shardingRouter.route(logicSQL, Collect
    }
}
```

可以看到在 StatementRoutingEngine 的 route 方法中，通过 ShardingMasterSlaveRouter 对通过 ShardingRouter 所生成的 SQLRouteResult 进行了再一次路由，也就是说在分片路由的基础上添加了主从路由，关于读写分离和主从路由我们会在之后的《26 | 读写分离：普通主从架构和分片主从架构分别是如何实现的？》进行讨论。

现在我们来到 sharding-core-entry 工程，看看更上层的处理流程。整个 sharding-core-entry 工程只有三个类，即作为基类的 BaseShardingEngine 以及两个子类 PreparedQueryShardingEngine 和 SimpleQueryShardingEngine。我们先来看 BaseShardingEngine 类，它本质上是一个模板类，BaseShardingEngine 的 shard 方法如下所示：

```
public SQLRouteResult shard(final String sql, final List<Object> parameters)
     //调用模板方法准备参数
    List<Object> clonedParameters = cloneParameters(parameters);
    //执行路由
    SQLRouteResult result = executeRoute(sql, clonedParameters);
    //执行 SQL 转换（Convert）和改写（Rewrite）
        result.getRouteUnits().addAll(HintManager.isDatabaseShardingOnly() ?

        //省略日志记录

    return result;
}
```

在这里我们看到了 SQL 转换（Convert）和改写（Rewrite）的入口，这是路由引擎之外的执行流程，我们今天不做展开。上述代码与路由相关最核心的就是 executeRoute 方法，如下所示：

```
private SQLRouteResult executeRoute(final String sql, final List<Object> clon
 routingHook.start(sql);
    try {
         //调用模板方法执行路由并获取结果
        SQLRouteResult result = route(sql, clonedParameters);
```

```
            routingHook.finishSuccess(result, metaData.getTables());
            return result;
        } catch (final Exception ex) {
            routingHook.finishFailure(ex);
            throw ex;
        }
    }
```

这个方法的处理方式与 SQLParseEngine 的 parse 方法有着类似的代码结构，同样用到了 Hook 机制。

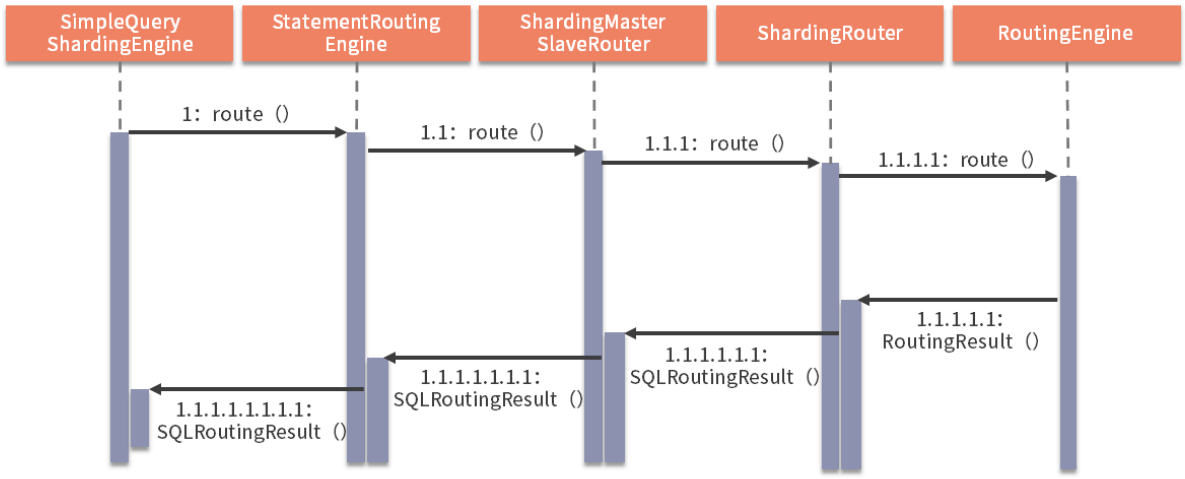
从设计模式上讲，BaseShardingEngine 采用了非常典型的模板方法。当我们需要完成一个过程或一系列步骤时，这些过程或步骤在某一细节层次保持一致，但个别步骤在更详细的层次上的实现可能不同时，可以考虑用模板方法模式来处理。实现模板方法的过程也非常简单，其实就是利用了类的继承机制。作为一个模板类，我们注意到 BaseShardingEngine 提供了两个模板方法供子类进行实现，分别是：

```
//拷贝参数
protected abstract List<Object> cloneParameters(List<Object> parameters);
//执行路由
protected abstract SQLRouteResult route(String sql, List<Object> parameters);
```

显然，对于 SimpleQueryShardingEngine 而言，不需要参数，所以 cloneParameters 直接返回空列表。而 route 方法则直接使用前面介绍的 StatementRoutingEngine 进行路由。SimpleQueryShardingEngine 类的完整实现如下所示：

```
public final class SimpleQueryShardingEngine extends BaseShardingEngine {

    private final StatementRoutingEngine routingEngine;

    public SimpleQueryShardingEngine(final ShardingRule shardingRule, final S
        super(shardingRule, shardingProperties, metaData);
        routingEngine = new StatementRoutingEngine(shardingRule, metaData, sq
    }

    @Override
    protected List<Object> cloneParameters(final List<Object> parameters) {
        return Collections.emptyList();
    }

    @Override
    protected SQLRouteResult route(final String sql, final List<Object> param
        return routingEngine.route(sql);
    }
}
```

至此，关于 ShardingSphere 路由引擎部分的内容基本都介绍完毕。对于上层结构而言，我们以 SimpleQueryShardingEngine 为例进行了展开，对于 PreparedQueryShardingEngine 的处理方式也是类似。作为总结，我们通过如下所示的时序图来梳理这些路由的主流程。



SimpleQueryShardingEngine 运作机制时序图

从源码解析到日常开发

分包设计原则可以用来设计和规划开源框架的代码结构。在今天的内容中，我们看到了 ShardingSphere 中非常典型的一种分层和分包实现策略。通过 sharding-core-route 和 sharding-core-entry 这两个工程，我们把路由引擎中位于底层的核心类 ShardingRouter 和位于上层的 PreparedQueryShardingEngine 及 SimpleQueryShardingEngine 类进行了合理的分层管理。ShardingSphere 对于分层和分包策略的应用有很多具体的表现形式，随着课程的不断演进，我们还会看到更多的应用场景。

小结与预告

作为 ShardingSphere 分片引擎的第二个核心组件，路由引擎的目的在于生成 SQLRouteResult目标对象。而整个路由引擎中最核心的就是 ShardingRouter 类。今天，我们对 ShardingRouter 的整体执行流程进行了详细的讨论，同时也引出了路由引擎中的底层对象 RoutingEngine。

这里给你留一道思考题：ShardingSphere 中，一个完整的路由执行过程需要经历哪些步骤？ 欢迎你在留言区与大家讨论，我将一一点评解答。

在今天的课程中，我们也提到了 ShardingSphere 中存在多种 RoutingEngine。在下一课时的内容中，我们将关注于这些 RoutingEngine 的具体实现过程。

上一页 下一页