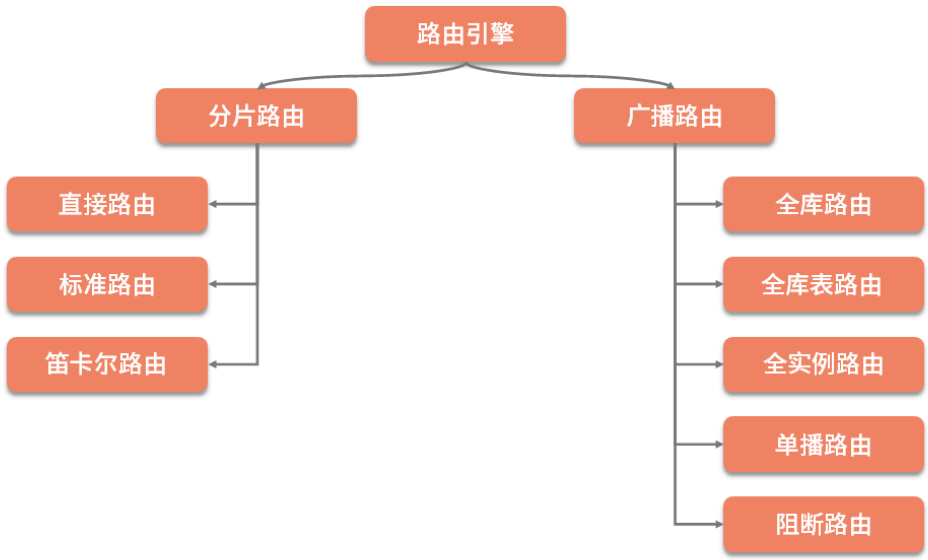
服务器出问题，目前部分恢复

18 路由引擎：如何实现数据访问的分片路由和广播路由？

在上一课时中，我们看到起到承上启下作用的 ShardingRouter 会调用 RoutingEngine 获取路由结果，而在 ShardingSphere 中存在多种不同类型的 RoutingEngine，分别针对不同的应用场景。

我们可以按照**是否携带分片键信息**将这些路由方式分成两大类，即分片路由和广播路由，而这两类路由中又存在一些常见的 RoutingEngine 实现类型，如下图所示：



RoutingEngine 实现的不同类型图

我们无意对所有这些 RoutingEngine 进行详细 的 展开，但在接下来的内容中，我们会分别对分片路由和广播路由中具有代表性的 RoutingEngine 进行讨论。

分片路由

对于分片路由而言，我们将重点介绍**标准路由**，标准路由是 ShardingSphere 推荐使用的分片方式。

在使用过程中，我们需要首先考虑标准路由的适用范围。标准路由适用范围有两大场景：一种面向不包含关联查询的 SQL；另一种则适用于仅包含绑定表关联查询的 SQL。前面一种场景比较好理解，而针对后者，我们就需要引入绑定表这个 ShardingSphere 中的重要概念。

关于绑定表，我们已经在 [《06 | 数据分片：如何实现分库、分表、分库+分表以及强制路由（上）？》]中进行了讨论，在明确了这些概念之后，我们来看标准路由的具体实现过程。

1.StandardRoutingEngine 的创建过程

明确了标准路由的基本含义之后，我们回顾一下上一课时中介绍的工厂类 RoutingEngineFactory。RoutingEngineFactory 类根据上下文中的路由信息构建对应的 RoutingEngine，但在其 newInstance 方法中，我们并没有发现直接创建 StandardRoutingEngine 的代码。事实上，StandardRoutingEngine 的创建是在 newInstance 方法中的最后一个代码分支，即当所有前置的判断都不成立时会进入到最后的 getShardingRoutingEngine 代码分支中，如下所示：

```
private static RoutingEngine getShardingRoutingEngine(final ShardingRule shar
                                                       final ShardingCondition
    //根据分片规则获取分片表
 Collection<String> shardingTableNames = shardingRule.getShardingLogicTableNa
    //如果目标表只要一张，或者说目标表都是绑定表关系，则构建StandardRoutingEngine
    if (1 == shardingTableNames.size() || shardingRule.isAllBindingTables(sha
        return new StandardRoutingEngine(shardingRule, shardingTableNames.ite
    }
    //否则构建ComplexRoutingEngine
    return new ComplexRoutingEngine(shardingRule, tableNames, sqlStatementCon
}
```

这段代码首先根据解析出来的逻辑表获取分片表，以如下所示的 SQL 语句为例：

```
SELECT record.remark_name FROM health_record record JOIN health_task task ON
```

那么 shardingTableNames 应该为 health_record 和 health_task。如果分片操作只涉及一张表，或者涉及多张表，但这些表是互为绑定表的关系时，则使用 StandardRoutingEngine 进行路由。

基于绑定表的概念，当多表互为绑定表关系时，每张表的路由结果是相同的，所以只要计算第一张表的分片即可；反之，如果不满足这一条件，则构建一个 ComplexRoutingEngine 进行路由。

这里我们来看一下代码中的 isAllBindingTables 方法如何对多表互为绑定表关系进行判定，该方法位于 ShardingRule 中，如下所示：

```
public boolean isAllBindingTables(final Collection<String> logicTableNames) {
    if (logicTableNames.isEmpty()) {
        return false;
    }
    //通过传入的logicTableNames构建一个专门的BindingTableRule
    Optional<BindingTableRule> bindingTableRule = findBindingTableRule(logicT
    if (!bindingTableRule.isPresent()) {
```

```
            return false;
        }
        Collection<String> result = new TreeSet<>(String.CASE_INSENSITIVE_ORDER);
        //获取BindingTableRule中的LogicTable
        result.addAll(bindingTableRule.get().getAllLogicTables());
        //判断获取的LogicTable是否与传入的logicTableNames一致
        return !result.isEmpty() && result.containsAll(logicTableNames);
    }
```

这段代码会通过传入的 logicTableNames 构建一个专门的 BindingTableRule，然后看最终获取的 BindingTableRule 中的 LogicTable 是否与传入的 logicTableNames 一致。这里构建 BindingTableRule 的过程实际上是根据传入的 logicTableName 来从 ShardingRule 中自身保存的 Collection <BindingTableRule> 获取对应的 BindingTableRule，如下所示：

```
public Optional<BindingTableRule> findBindingTableRule(final String logicTabl
        for (BindingTableRule each : bindingTableRules) {
            if (each.hasLogicTable(logicTableName)) {
                return Optional.of(each);
            }
        }
        return Optional.absent();
    }
```

上述代码的 bindingTableRules 就是 ShardingRule 中自身保存的 BindingTableRule 集合，我们在 ShardingRule 构造函数中发现了初始化 bindingTableRules 的代码，如下所示：

```
bindingTableRules = createBindingTableRules(shardingRuleConfig.getBindingTabl
```

显然，这个构建过程与规则配置机制有关。如果基于 Yaml 配置文件，绑定表的配置一般会采用如下形式：

```
shardingRule:
  bindingTables:
     health_record,health_task
```

针对这种配置形式，ShardingRule 会对其进行解析并生成 BindingTableRule 对象，如下所示：

```
private BindingTableRule createBindingTableRule(final String bindingTableGrou
    List<TableRule> tableRules = new LinkedList<>();
    for (String each : Splitter.on(",").trimResults().splitToList(bindingTabl
        tableRules.add(getTableRule(each));
    }
    return new BindingTableRule(tableRules);
}
```

至此，我们终于把绑定表相关的概念以及实现方式做了介绍，也就是说完成了 RoutingEngineFactory 中进入到 StandardRoutingEngine 这条代码分支的介绍。

2.StandardRoutingEngine 的运行机制

现在，我们已经创建了 StandardRoutingEngine，接下来就看它的运行机制。作为一种具体的路由引擎实现方案，StandardRoutingEngine 实现了 RoutingEngine 接口，它的 route 方法如下所示：

```
@Override
public RoutingResult route() {
        …
        return generateRoutingResult(getDataNodes(shardingRule.getTableRule(l
}
```

这里的核心方法就是 generateRoutingResult，在此之前需要先通过 getDataNodes 方法来获取数据节点信息，该方法如下所示：

```
private Collection<DataNode> getDataNodes(final TableRule tableRule) {
     //如基于Hint进行路由
    if (isRoutingByHint(tableRule)) {
        return routeByHint(tableRule);
    }
    //基于分片条件进行路由
    if (isRoutingByShardingConditions(tableRule)) {
        return routeByShardingConditions(tableRule);
    }
    //执行混合路由
    return routeByMixedConditions(tableRule);
}
```

我们看到这个方法的入参是一个 TableRule 对象，而 TableRule 属于分片规则 ShardingRule 中的一部分。我们在上一课时中知道该对象主要保存着与分片相关的各种规则信息，其中就包括 ShardingStrategy。从命名上看，ShardingStrategy 属于一种分片策略，用于指定分片的具体 Column，以及执行分片并返回目标 DataSource 和 Table。

这部分内容我们会在下一课时中进行展开。这里，我们先梳理与 ShardingStrategy 相关的类结构，如下所示：

![image(assets/CgqCHl81FfKAYWCOAACN0o0OVu8479.png)

在 StandardRoutingEngine 中，整体结构也与上图类似。在 StandardRoutingEngine 中，前面所介绍的 getDataNodes 方法的第一个判断分支 isRoutingByHint 方法中会判断是否根据 Hint 来进行路由，其判断依据是它的 DatabaseShardingStrategy 和 TableShardingStrategy 是否都为 HintShardingStrategy，这个方法就用到了 ShardingRule 的这两个 ShardingStrategy 对象，如下所示：

```
private boolean isRoutingByHint(final TableRule tableRule) {
    return shardingRule.getDatabaseShardingStrategy(tableRule) instanceof Hin
}
```

在 ShardingSphere 中，Hint 代表的是一种强制路由的方法，是一条流程的支线。然后，我们再看 getDataNodes 方法中的 isRoutingByShardingConditions 判断。想要判断是否根据分片条件进行路由，其逻辑在于 DatabaseShardingStrategy 和 TableShardingStrategy 都不是 HintShardingStrategy 时就走这个代码分支。而最终如果 isRoutingByHint 和 isRoutingByShardingConditions 都不满足，也就是说，DatabaseShardingStrategy 或 TableShardingStrategy 中任意一个是 HintShardingStrategy，则执行 routeByMixedConditions 这一混合的路由方式。

以上三条代码分支虽然处理方式有所不同，但**本质上都是获取 RouteValue 的集合**，我们在上一课时中介绍路由条件 ShardingCondition 时知道，RouteValue 保存的就是用于路由的表名和列名。在获取了所需的 RouteValue 之后，在 StandardRoutingEngine 中，以上三种场景最终都会调用 route0 基础方法进行路由，该方法的作用就是根据这些 RouteValue 得出目标 DataNode 的集合。同样，我们也知道 DataNode 中保存的就是具体的目标节点，包括 dataSourceName和tableName。route0 方法如下所示：

```
private Collection<DataNode> route0(final TableRule tableRule, final List<Rou
    //路由DataSource
 Collection<String> routedDataSources = routeDataSources(tableRule, databaseS
    Collection<DataNode> result = new LinkedList<>();
    //路由Table，并完成DataNode集合的拼装
    for (String each : routedDataSources) {
        result.addAll(routeTables(tableRule, each, tableShardingValues));
    }
    return result;
}
```

可以看到，该方法首先路由 DataSource，然后再根据每个 DataSource 路由 Table，最终完成 DataNode 集合的拼装。在上述 routeDataSources 和 routeTables 方法中，最终都会分别依赖 DatabaseShardingStrategy 和 TableShardingStrategy 完成背后的路由计算以获取目标 DataSource 以及 Table。

当获取了 DataNode 集合之后，我们回到 StandardRoutingEngine 的 generateRoutingResult 方法，该方法用于组装路由结果并返回一个 RoutingResult：

```
private RoutingResult generateRoutingResult(final Collection<DataNode> routed
    RoutingResult result = new RoutingResult();
    for (DataNode each : routedDataNodes) {
     //根据每个DataNode构建一个RoutingUnit对象
        RoutingUnit routingUnit = new RoutingUnit(each.getDataSourceName());
        //填充RoutingUnit中的TableUnit
        routingUnit.getTableUnits().add(new TableUnit(logicTableName, each.ge
        result.getRoutingUnits().add(routingUnit);
    }
```

```
        return result;
    }
```

这部分代码的作用就是根据每个 DataNode 构建一个 RoutingUnit 对象，然后再填充 RoutingUnit 中的 TableUnit。关于 RoutingUnit 和 TableUnit 的数据结构我们在上一课时中已经进行了介绍，这里不再展开。

至此，对标准路由引擎 StandardRoutingEngine 的介绍就告一段落，标准路由是 ShardingSphere 最为推荐使用的分片方式，在日常开发中应用也最广泛。

广播路由

对于不携带分片键的 SQL，路由引擎会采取广播路由的方式。在 ShardingSphere，根据输入 SQL 的类型，存在很多种用于广播的路由引擎，我们同样可以回顾 RoutingEngineFactory 中创建 RoutingEngine的 方法。

首先，如果输入的是 TCLStatement，即授权、角色控制等数据库控制语言，那么直接执行 DatabaseBroadcastRoutingEngine；同样，如果执行的是用于数据定义的 DDLStatement，则执行 TableBroadcastRoutingEngine 中的路由方法，判断条件如下所示：

```
//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.getTables()
}
```

DatabaseBroadcastRoutingEngine 的路由方法非常直接，即基于每个 DataSourceName 构建一个 RoutingUnit，然后再拼装成 RoutingResult，如下所示：

```
public final class DatabaseBroadcastRoutingEngine implements RoutingEngine {

    private final ShardingRule shardingRule;

    @Override
    public RoutingResult route() {
        RoutingResult result = new RoutingResult();
        for (String each : shardingRule.getShardingDataSourceNames().getDataS
         //基于每个DataSourceName构建一个RoutingUnit
            result.getRoutingUnits().add(new RoutingUnit(each));
        }
        return result;
    }
}
```

同样也可以想象 TableBroadcastRoutingEngine 的实现过程，我们根据 logicTableName 获取对应的 TableRule，然后根据 TableRule 中的真实 DataNode 构建 RoutingUnit 对象，这一过程如下所示：

```
private Collection<RoutingUnit> getAllRoutingUnits(final String logicTableNam
    Collection<RoutingUnit> result = new LinkedList<>();
    //根据logicTableName获取对应的TableRule
    TableRule tableRule = shardingRule.getTableRule(logicTableName);
    for (DataNode each : tableRule.getActualDataNodes()) {
        //根据TableRule中的真实DataNode构建RoutingUnit对象
        RoutingUnit routingUnit = new RoutingUnit(each.getDataSourceName());
        //根据DataNode的TableName构建TableUnit
        routingUnit.getTableUnits().add(new TableUnit(logicTableName, each.ge
        result.add(routingUnit);
    }
    return result;
}
```

接着我们来看针对 DALStatement 的场景，这一场景相对复杂，根据输入的 DALStatement 的不同类型，会有几个不同的处理分支，如下所示：

```
private static RoutingEngine getDALRoutingEngine(final ShardingRule shardingR
         //如果是Use语句，则什么也不做
        if (sqlStatement instanceof UseStatement) {
            return new IgnoreRoutingEngine();
        }
         //如果是Set或ResetParameter语句，则进行全数据库广播
        if (sqlStatement instanceof SetStatement || sqlStatement instanceof R
            return new DatabaseBroadcastRoutingEngine(shardingRule);
        }
        //如果存在默认数据库，则执行默认数据库路由
        if (!tableNames.isEmpty() && !shardingRule.tableRuleExists(tableNames
            return new DefaultDatabaseRoutingEngine(shardingRule, tableNames)
        }
        //如果表列表不为空，则执行单播路由
        if (!tableNames.isEmpty()) {
            return new UnicastRoutingEngine(shardingRule, tableNames);
        }
        //
        return new DataSourceGroupBroadcastRoutingEngine(shardingRule);
}
```

我们分别来看一下这里面的几个路由引擎。首先是最简单的 IgnoreRoutingEngine，它只返回一个空的 RoutingResult 对象，其他什么都不做，如下所示：

```
public final class IgnoreRoutingEngine implements RoutingEngine {

    @Override
    public RoutingResult route() {
        return new RoutingResult();
```

```
    }
}
```

本质上，UnicastRoutingEngine 代表单播路由，用于获取某一真实表信息的场景，它只需要从任意库中的任意真实表中获取数据即可。例如 DESCRIBE 语句就适合使用 UnicastRoutingEngine，因为每个真实表中的数据描述结构都是相同的。

UnicastRoutingEngine 实现过程如下所示，由于方法比较长，我们裁剪了代码，直接使用注释来标明每个分支的执行逻辑：

```
@Override
public RoutingResult route() {
    RoutingResult result = new RoutingResult();
    if (shardingRule.isAllBroadcastTables(logicTables)) {
        //如果都是广播表，则对每个logicTable组装TableUnit，再构建RoutingUnit
    } else if (logicTables.isEmpty()) {
        //如果表为null，则直接组装RoutingUnit，不用构建TableUnit
    } else if (1 == logicTables.size()) {
        //如果只有一张表，则组装RoutingUnit和单个表的TableUnit
    } else {
        //如果存在多个实体表，则先获取DataSource，再组装RoutingUnit和TableUnit
    }
    return result;
}
```

DefaultDatabaseRoutingEngine，顾名思义是对默认的数据库执行路由。那么这个默认数据库是怎么来的呢？我们可以从 ShardingRule的ShardingDataSourceNames 类中的 getDefaultDataSourceName 方法中找到答案。

一般，这种默认数据库可以通过配置的方式进行设置。明白这一点，DefaultDatabaseRoutingEngine 的路由过程也就不难理解了，其 route 方法如下所示：

```
@Override
public RoutingResult route() {
    RoutingResult result = new RoutingResult();
    List<TableUnit> routingTables = new ArrayList<>(logicTables.size());
    for (String each : logicTables) {
        routingTables.add(new TableUnit(each, each));
    }
    //从ShardingRule中获取默认所配置的数据库名
    RoutingUnit routingUnit = new RoutingUnit(shardingRule.getShardingDataSou
    routingUnit.getTableUnits().addAll(routingTables);
    result.getRoutingUnits().add(routingUnit);
    return result;
}
```

最后，我们来看一下针对数据控制语言 DCLStatement 的处理流程。在主从环境下，对于 DCLStatement 而言，有时候我们希望 SQL 语句只针对主数据库进行执行，所以就有了如下所示的 MasterInstanceBroadcastRoutingEngine：

```
@Override
public RoutingResult route() {
    RoutingResult result = new RoutingResult();
    for (String each : shardingRule.getShardingDataSourceNames().getDataSourc
        if (dataSourceMetas.getAllInstanceDataSourceNames().contains(each)) {
          //通过MasterSlaveRule获取主从数据库信息
            Optional<MasterSlaveRule> masterSlaveRule = shardingRule.findMast
            if (!masterSlaveRule.isPresent() || masterSlaveRule.get().getMast
                result.getRoutingUnits().add(new RoutingUnit(each));
            }
        }
    }
    return result;
}
```

可以看到，这里引入了一个 MasterSlaveRule 规则，该规则提供 getMasterDataSourceName 方法以获取主 DataSourceName，这样我们就可以针对这个主数据执行，如 Grant 等数据控制语言。

从源码解析到日常开发

在 ShardingSphere 中，我们还是有必要再次强调其在配置信息管理上的一些设计和实践。基于 ShardingRule 和 TableRule 这两个配置类，ShardingSphere 把大量纷繁复杂的配置信息从业务流程中进行隔离，而这些配置信息往往需要灵活进行设置，以及多种默认配置值。基于 ShardingRule 和 TableRule 的两层配置体系，系统能够更好地完成业务逻辑的变化和配置信息变化之间的有效整合，值得我们在日常开发过程中进行尝试和应用。

小结与预告

今天我们关注的是 ShardingSphere 中各种路由引擎的实现过程，ShardingSphere 中实现了多款不同的路由引擎，可以分为分片路由和广播路由两大类。我们针对这两类路由引擎中的代表性实现方案分别展开了讨论。

这里给你留一道思考题：ShardingSphere 中如何判断两张表是互为绑定表关系？ 欢迎你在留言区与大家讨论，我将一一点评解答。

从今天的内容中，我们也看到了路由引擎中路由机制的实现需要依赖于分片策略及其背后分片算法的集成，下一课时将对 ShardingSphere 中的各种分片策略进行具体的展开。

上一页 下一页