芋艿v的博客

愿编码半生，如老友相伴！

分享

微信公众号福利：芋艿的后端小屋

0. 阅读源码葵花宝典

1. RocketMQ / MyCAT / Sharding-JDBC 详细中文注释源码

2. 您对于源码的疑问每条留言都将得到认真回复

3. 新的源码解析文章实时收到通知，每周六十点更新

4. 认真的源码交流微信群

分类

分享

Sharding-JDBC 源码分析 —— SQL 路由（一）之分库分表配置

2017-08-04 更新日期:2017-08-02 总阅读量:23次

文章目录

> □□□关注**微信公众号：【芋艿的后端小屋】**有福利：
>
> 1. RocketMQ / MyCAT / Sharding-JDBC **所有**源码分析文章列表
> 2. RocketMQ / MyCAT / Sharding-JDBC **中文注释源码 GitHub 地址**
> 3. 您对于源码的疑问每条留言**都**将得到**认真**回复。**甚至不知道如何读源码也可以请教噢**。
> 4. **新的**源码解析文章**实时**收到通知。**每周更新一篇左右**。
> 5. **认真的**源码交流微信群。

---

1. 概述

😁《SQL 解析》已经告于段落，我们要开始新的旅程：《SQL 路由》。相比**SQL解析**，路由会容易理解很多，骗人是小☎。整个系列预计会拆分成**三小篇**文章：

1. 《分库分表配置》
2. 《分表分库路由》

3. 《Spring与YAML配置》

第一、二篇会在**近期**更新。第三篇会在《SQL 改写》、《SQL 执行》完成后进行更新。😈改写和执行相对有趣。

🚶道友，您看，逗比博主**“很有规划”**，是关注公众号一波【芋艿的后端小屋】还是分享朋友圈。

阅读本文之前，建议已经读过**官方**相关文章：

- 《Sharding-JDBC 核心概念》
- 《Sharding-JDBC 分表分库》

分表分库配置会涉及如下类：

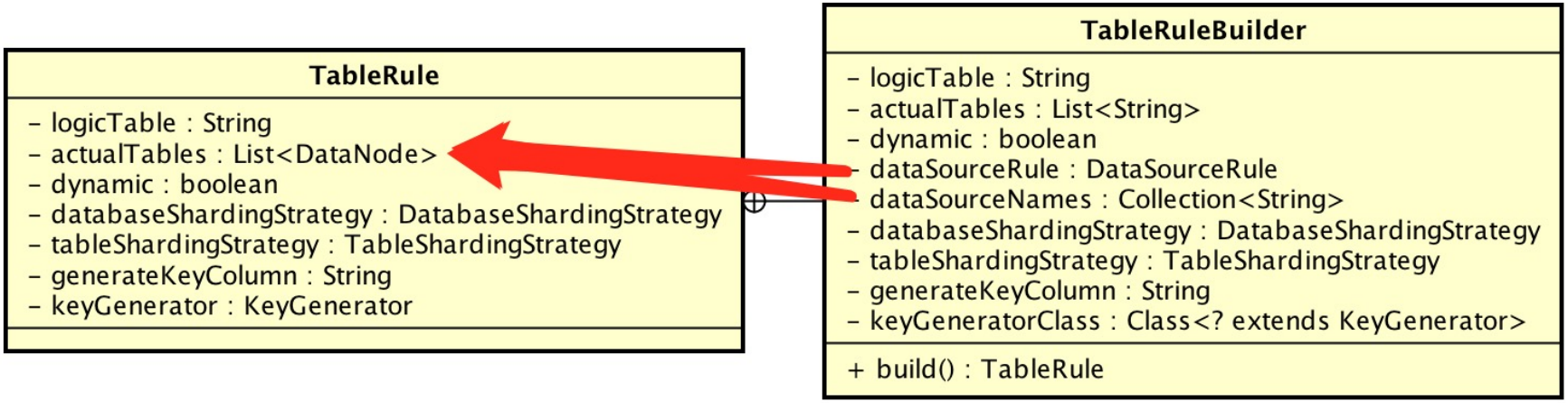
- TableRule 表规则配置对象
- ShardingRule 分库分表规则配置对象
- ShardingStrategy 分片策略
- ShardingAlgorithm 分片算法

我们来一起逐个类往下看。

> **Sharding-JDBC 正在收集使用公司名单：传送门。**
> **□ 你的登记，会让更多人参与和使用 Sharding-JDBC。传送门**
> **Sharding-JDBC 也会因此，能够覆盖更多的业务场景。传送门**
> **登记吧，骚年！传送门**

2. TableRule

TableRule，表规则配置对象，内嵌 TableRuleBuilder 对象进行创建。



2.1 logicTable

> 数据分片的**逻辑表**，对于水平拆分的数据库(表)，同一类表的总称。
> 例：订单数据根据主键尾数拆分为10张表,分别是t_order_0到t_order_9，他们的逻辑表名为t_order。

2.2 数据单元

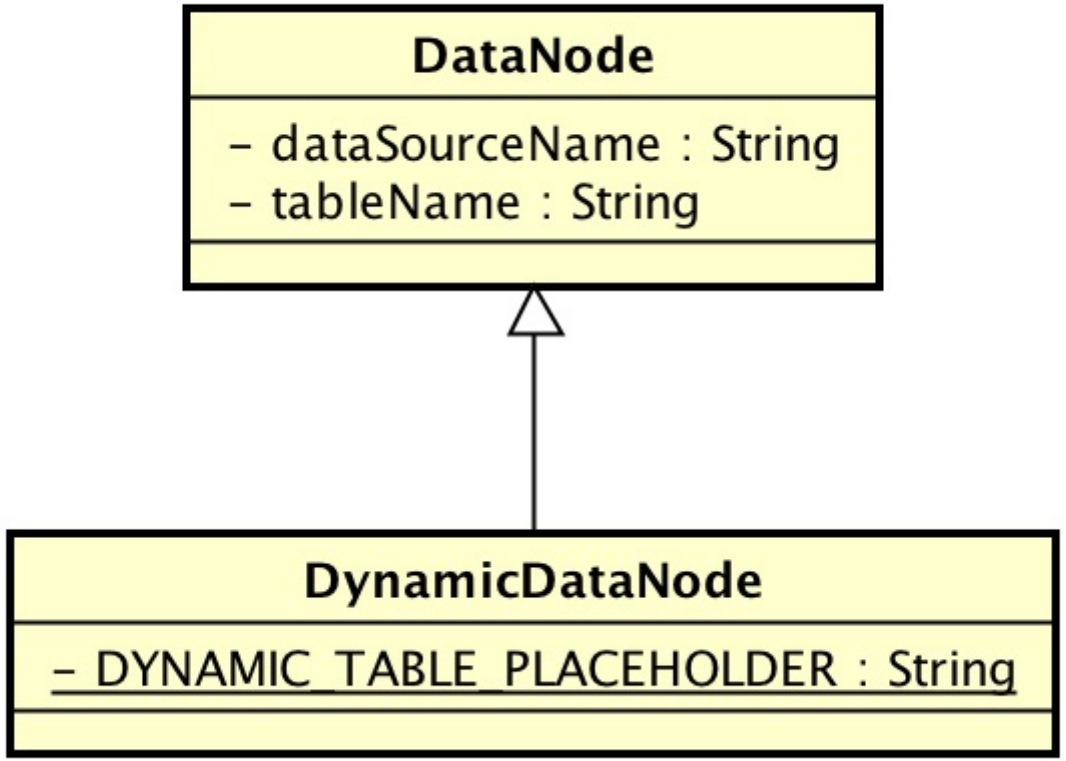
Sharding-JDBC 有两种类型**数据单元**：

- DataNode ：**静态**分库分表数据单元

> 数据分片的最小单元，由数据源名称和数据表组成。
> 例：ds_1.t_order_0。配置时默认各个分片数据库的表结构均相同，直接配置逻辑表和真实表对应关系即可。
> 如果各数据库的表结果不同，可使用ds.actual_table配置。

- DynamicDataNode ：**动态**表的分库分表数据单元

> 逻辑表和真实表不一定需要在配置规则中静态配置。
> 比如按照日期分片的场景，真实表的名称随着时间的推移会产生变化。
> 此类需求Sharding-JDBC是支持的，不过目前配置并不友好，会在新版本中提升。



TableRuleBuilder 调用 `#build()` 方法创建 TableRule。核心代码如下：

```
// TableRuleBuilder.java
public static class TableRuleBuilder {
  public TableRule build() {
        KeyGenerator keyGenerator = null;
        if (null != generateKeyColumn && null != keyGeneratorClass) {
            keyGenerator = KeyGeneratorFactory.createKeyGenerator(keyGeneratorClass);
        }
        return new TableRule(logicTable, dynamic, actualTables, dataSourceRule, dataSourceNames, databa
```

```
    }
}
// TableRule.java
public TableRule(final String logicTable, final boolean dynamic, final List<String> actualTables, fina
                 final DatabaseShardingStrategy databaseShardingStrategy, final TableShardingStrategy t
                 final String generateKeyColumn, final KeyGenerator keyGenerator) {
    Preconditions.checkNotNull(logicTable);
    this.logicTable = logicTable;
    this.dynamic = dynamic;
    this.databaseShardingStrategy = databaseShardingStrategy;
    this.tableShardingStrategy = tableShardingStrategy;
    if (dynamic) { // 动态表的分库分表数据单元
        Preconditions.checkNotNull(dataSourceRule);
        this.actualTables = generateDataNodes(dataSourceRule);
    } else if (null == actualTables || actualTables.isEmpty()) { // 静态表的分库分表数据单元
        Preconditions.checkNotNull(dataSourceRule);
        this.actualTables = generateDataNodes(Collections.singletonList(logicTable), dataSourceRule, da
    } else { // 静态表的分库分表数据单元
        this.actualTables = generateDataNodes(actualTables, dataSourceRule, dataSourceNames);
    }
    this.generateKeyColumn = generateKeyColumn;
    this.keyGenerator = keyGenerator;
}
```

2.2.1 DataNode

大多数业务场景下，我们使用**静态**分库分表数据单元，即 DataNode。如上文注释处 `静态表的分库分表数据单元` 处所见，分成**两**种判断，实质上第一种是将 `logicTable` 作为 `actualTable` ，即在**库**里不进行分表，是第二种的一种特例。

我们来看看 `#generateDataNodes()` 方法：

```
// TableRule.java
/**
* 生成静态数据分片节点
*
* @param actualTables 真实表
* @param dataSourceRule 数据源配置对象
* @param actualDataSourceNames 数据源名集合
* @return 静态数据分片节点
*/
private List<DataNode> generateDataNodes(final List<String> actualTables, final DataSourceRule dataSou
   Collection<String> dataSourceNames = getDataSourceNames(dataSourceRule, actualDataSourceNames);
   List<DataNode> result = new ArrayList<>(actualTables.size() * (dataSourceNames.isEmpty() ? 1 : data
   for (String actualTable : actualTables) {
       if (DataNode.isValidDataNode(actualTable)) { // 当 actualTable 为 ${dataSourceName}.${tableName
           result.add(new DataNode(actualTable));
       } else {
           for (String dataSourceName : dataSourceNames) {
               result.add(new DataNode(dataSourceName, actualTable));
           }
       }
   }
   return result;
}
/**
```

分享

```
     * 根据 数据源配置对象 和 数据源名集合 获得 最终的数据源名集合
     *
     * @param dataSourceRule 数据源配置对象
     * @param actualDataSourceNames 数据源名集合
     * @return 最终的数据源名集合
     */
    private Collection<String> getDataSourceNames(final DataSourceRule dataSourceRule, final Collection<St
        if (null == dataSourceRule) {
            return Collections.emptyList();
        }
        if (null == actualDataSourceNames || actualDataSourceNames.isEmpty()) {
            return dataSourceRule.getDataSourceNames();
        }
        return actualDataSourceNames;
    }
```

- 第一种情况，**自定义分布**。`actualTable` 为 `${dataSourceName}.${tableName}` 时，即已经明确真实表所在数据源。

```
TableRule.builder("t_order").actualTables(Arrays.asList("db0.t_order_0", "db1.t_order_1", "db1.t_order
```

```
db0
  └── t_order_0
db1
  ├── t_order_1
  └── t_order_2
```

- 第二种情况，**均匀分布**。

```
TableRule.builder("t_order").actualTables(Arrays.asList("t_order_0", "t_order_1"))
```

```
db0
  ├── t_order_0
  └── t_order_1
db1
  ├── t_order_0
  └── t_order_1
```

`#getDataSourceNames()` 使用 `dataSourceRule` 和 `actualDataSourceNames` 获取数据源的逻辑看起来有种“诡异”。**实际 TableRuleBuilder 创建 TableRule 时，使用 `dataSourceRule` 而不要使用 `actualDataSourceNames`**。

2.2.2 DynamicDataNode

少数业务场景下，我们使用**动态**分库分表数据单元，即 DynamicDataNode。 **通过 `dynamic=true` 属性配置**。生成代码如下：

```
// TableRule.java
private List<DataNode> generateDataNodes(final DataSourceRule dataSourceRule) {
    Collection<String> dataSourceNames = dataSourceRule.getDataSourceNames();
    List<DataNode> result = new ArrayList<>(dataSourceNames.size());
    for (String each : dataSourceNames) {
        result.add(new DynamicDataNode(each));
    }
    return result;
```

```
}
```

😂 从代码上看，貌似和**动态**分库分表数据单元没一毛钱关系？！别捉鸡，答案在《分表分库路由》 上。

2.3 分库/分表策略

- `databaseShardingStrategy` ：分库策略
- `tableShardingStrategy` ：分表策略

当分库/分表策略不配置时，使用 ShardingRule 配置的分库/分表策略。

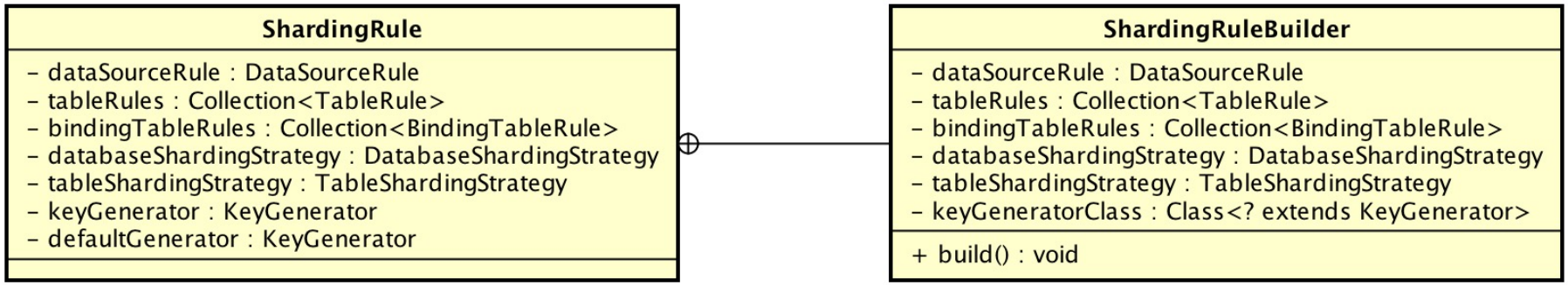
2.4 主键生成

- `generateKeyColumn` ：主键字段
- `keyGenerator` ：主键生成器

当主键生成器不配置时，使用 ShardingRule 配置的主键生成器。

3. ShardingRule

ShardingRule，分库分表规则配置对象，内嵌 ShardingRuleBuilder 对象进行创建。



其中 databaseShardingStrategy、tableShardingStrategy、keyGenerator、defaultGenerator 和 TableRule 属性重复，用于当 TableRule 未配置对应属性，使用 ShardingRule 提供的该属性。

3.1 dataSourceRule

`dataSourceRule` ，数据源配置对象。ShardingRule 需要数据源配置正确。这点和 TableRule 是不同的。TableRule 对 `dataSourceRule` **只使用数据源名字，最终执行SQL 使用数据源名字从 ShardingRule 获取数据源连接**。大家可以回到本文【2.2.1 DataNode】细看下 DataNode 的生成过程。

3.2 tableRules

`tableRules` ，表规则配置对象**集合**。

3.3 bindingTableRules

> 指在任何场景下分片规则均一致的主表和子表。
> 例：订单表和订单项表，均按照订单ID分片，则此两张表互为BindingTable关系。

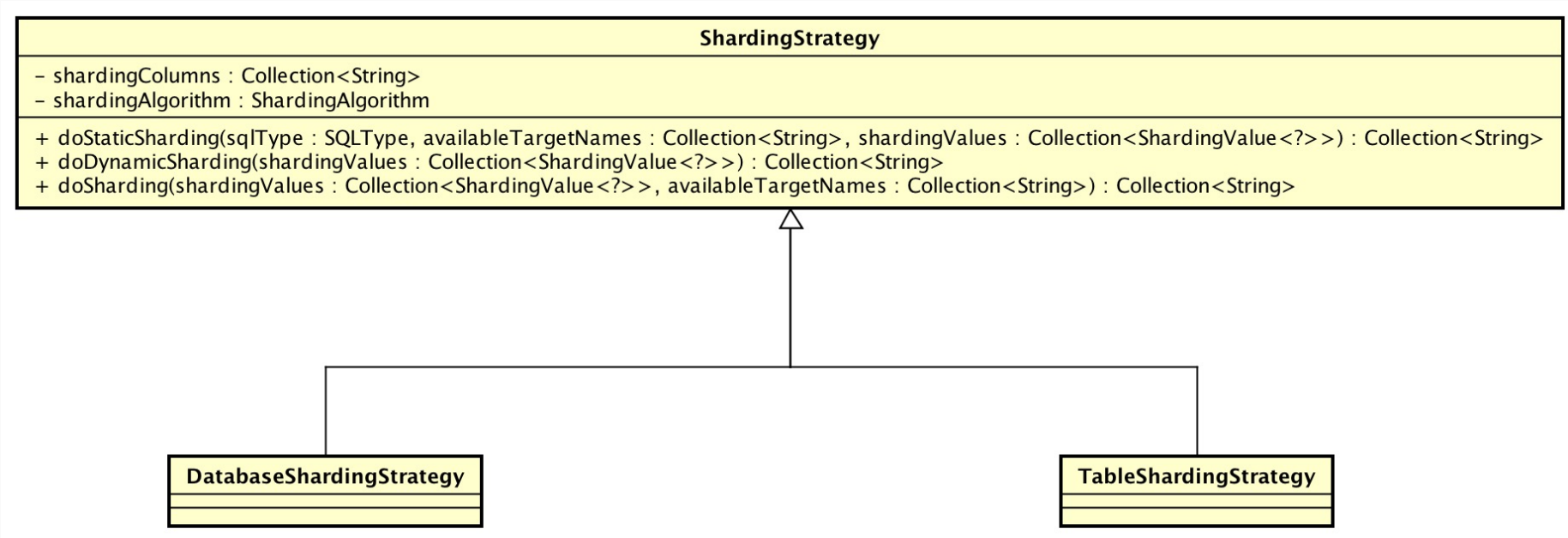
> BindingTable关系的多表关联查询不会出现笛卡尔积关联，关联查询效率将大大提升。

😺 这么说，可能不太容易理解。《分表分库路由》，我们在源码的基础上，好好理解下。**非常重要，特别是性能优化上面。**

4. ShardingStrategy

ShardingStrategy，分片策略。

- 针对分库、分表有两个子类。



- DatabaseShardingStrategy，使用**分库**算法进行分片
- TableShardingStrategy，使用**分表**算法进行分片

《分表分库路由》会进一步说明。

5. ShardingAlgorithm

ShardingAlgorithm，分片算法。

- 针对分库、分表有两个子**接口**。
- 针对**分片键**数量分成：无分片键算法、单片键算法、多片键算法。

其中 NoneKeyDatabaseShardingAlgorithm、NoneTableShardingAlgorithm 为 ShardingRule 在未设置分库、分表算法的默认值。代码如下：

```
// ShardingRule.java
public ShardingRule(
        final DataSourceRule dataSourceRule, final Collection<TableRule> tableRules, final Collection<B
        final DatabaseShardingStrategy databaseShardingStrategy, final TableShardingStrategy tableShard
    // ... 省略部分代码
    this.databaseShardingStrategy = null == databaseShardingStrategy ? new DatabaseShardingStrategy(
            Collections.<String>emptyList(), new NoneDatabaseShardingAlgorithm()) : databaseShardingStr
    this.tableShardingStrategy = null == tableShardingStrategy ? new TableShardingStrategy(
            Collections.<String>emptyList(), new NoneTableShardingAlgorithm()) : tableShardingStrategy;
    // ... 省略部分代码
}
```

《分表分库路由》会进一步说明。

666. 彩蛋

本文看似在水更，实是为《分表分库路由》做铺垫（一阵脸红😊）。

But，无论怎么说，道友，我做了新的关注二维码（感谢猫🐱先生），是不是可以推荐一波公众号给基佬。

恩，继续更新。

Sharding-JDBC

PREVIOUS:
« Sharding-JDBC 源码分析 —— SQL 路由（二）之分库分表路由
NEXT:
» Sharding-JDBC 源码分析 —— SQL 解析（六）之删除SQL

分享