课程目标

1、掌握 Sharding-JDBC 的使用方式，掌握自定义分片算法的配置

2、掌握分布式事务、全局 ID 等问题的解决方案

3、理解 Sharding-JDBC 的工作流程和实现原理

4、理解基于客户端的分库分表方案和基于代理的分库分表方案的差别

内容定位

适合已经理解了分库分表的意义、分库分表的类型，不知道如何实现在客户端实现分库分表的同学

1 架构与核心概念

https://gitee.com/Sharding-Sphere/sharding-sphere

1.1 回顾

数据源选择的解决方案层次：

DAO：AbstractRoutingDataSource

ORM：MyBatis 插件

JDBC：Sharding-JDBC

Proxy：Mycat、Sharding-Proxy

Server：特定数据库或者版本

1.2 发展历史

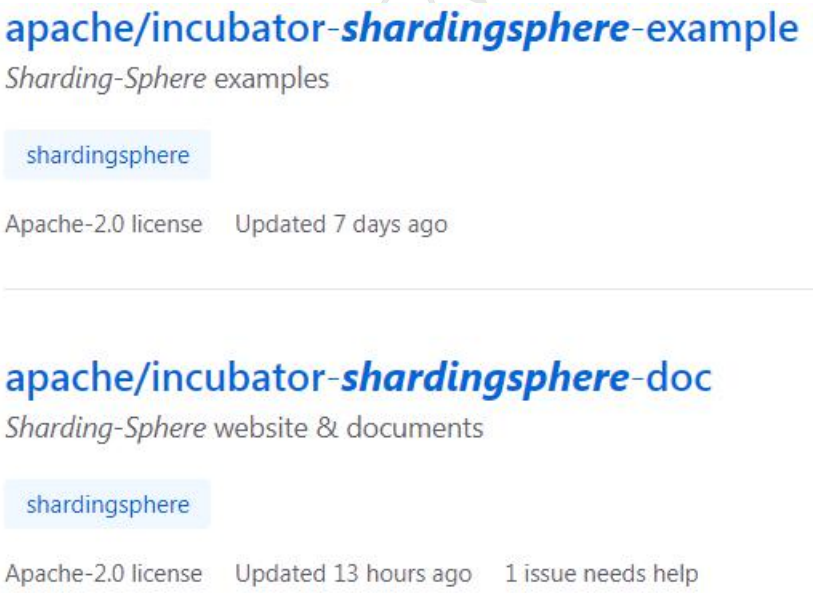
它是从当当网的内部架构 ddframe 里面的一个分库分表的模块脱胎出来的，用来解决当当的分库分表的问题，把跟业务相关的敏感的代码剥离后，就得到了 Sharding-JDBC。它是一个工作在客户端的分库分表的解决方案。

DubboX，Elastic-job 也是当当开源出来的产品。

2018 年 5 月，因为增加了 Proxy 的版本和 Sharding-Sidecar（尚未发布），Sharding-JDBC 更名为 Sharding Sphere，从一个客户端的组件变成了一个套件。

2018 年 11 月，Sharding-Sphere 正式进入 Apache 基金会孵化器，这也是对 Sharding-Sphere 的质量和影响力的认可。不过现在还没有毕业（名字带 incubator），一般我们用的还是 io.shardingsphere 的包。

现在 Sharding-Sphere 已经不属于当当网，也不属于作者张亮个人了。



因为更名后和捐献给 Apache 之后的 groupId 都不一样，在引入依赖的时候千万要注意。主体功能是相同的，但是在某些类的用法上有些差异，如果要升级的话 import 要全部修改，有些类和方法也要修改。

1.3 基本特性

Sharding-JDBC 是怎么工作的?

https://shardingsphere.apache.org/document/current/cn/overview/

我们看一下官网的定义：

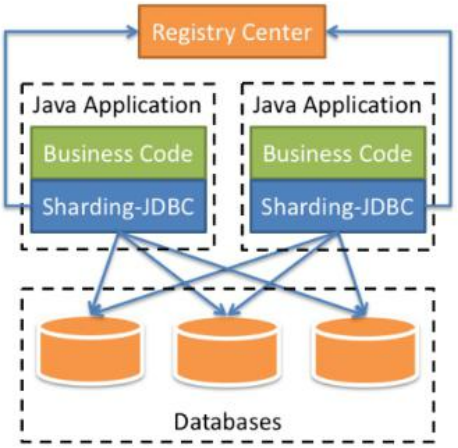
> 定位为轻量级 Java 框架，在 Java 的 JDBC 层提供的额外服务。 它使用客户端直连数据库，以 jar 包形式提供服务，无需额外部署和依赖，可理解为增强版的 JDBC 驱动，完全兼容 JDBC 和各种 ORM 框架。

在 maven 的工程里面，我们使用它的方式是引入依赖，然后进行配置就可以了，不用像 Mycat 一样独立运行一个服务，客户端不需要修改任何一行代码，原来是 SSM 连接数据库，还是 SSM，因为它是支持 MyBatis 的。

1.4 架构

我们在项目内引入 Sharding-JDBC 的依赖，我们的业务代码在操作数据库的时候，就会通过 Sharding-JDBC 的代码连接到数据库。

分库分表的一些核心动作，比如 SQL 解析，路由，执行，结果处理，都是由它来完成的。它工作在客户端。



在 Sharding-Sphere 里面同样提供了代理 Proxy 的版本，跟 Mycat 的作用是一样的。Sharding-Sidecar 是一个 Kubernetes 的云原生数据库代理，正在开发中。

	Sharding-JDBC	*Sharding-Proxy*	*Sharding-Sidecar*
数据库	任意	MySQL	MySQL
连接消耗数	高	低	高
异构语言	仅Java	任意	任意
性能	损耗低	损耗略高	损耗低
无中心化	是	否	是
静态入口	无	有	无

1.5 功能

分库分表后的几大问题：跨库关联查询、分布式事务、排序翻页计算、全局主键。

1.5.1 数据分片

1、分库 & 分表

2、读写分离

https://shardingsphere.apache.org/document/current/cn/features/read-write-split/

3、分片策略定制化

4、无中心化分布式主键（包括 UUID、雪花、LEAF）

https://shardingsphere.apache.org/document/current/cn/features/sharding/other-features/key-generator/

1.5.2 分布式事务

https://shardingsphere.apache.org/document/current/cn/features/transaction/

1、标准化事务接口

2、XA 强一致事务

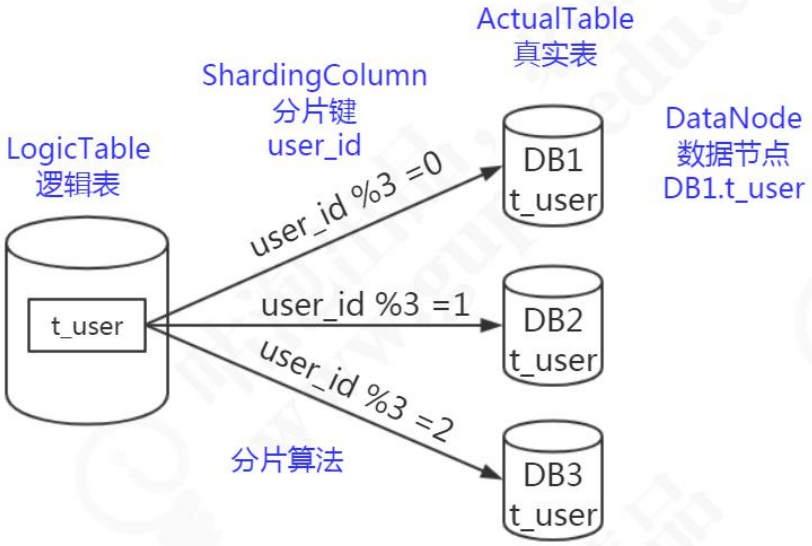
3、柔性事务

1.6 核心概念

https://shardingsphere.apache.org/document/current/cn/features/sharding/concept/sql/

逻辑表、真实表、分片键、数据节点、动态表、广播表、绑定表

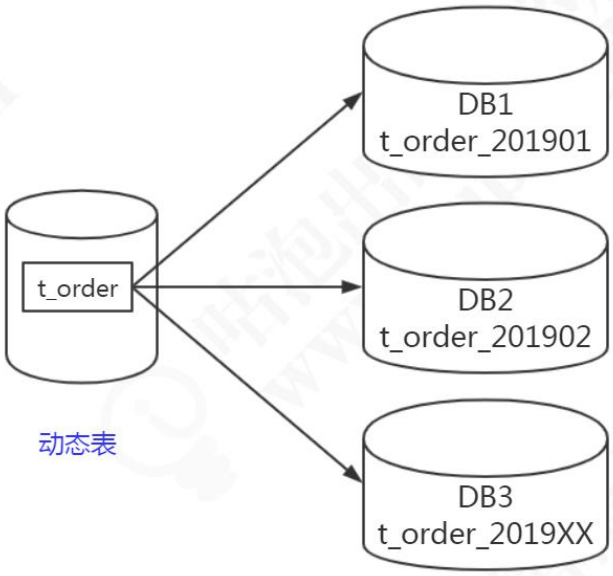
1.6.1 主要概念



逻辑表会在 SQL 解析和路由时被替换成真实的表名。

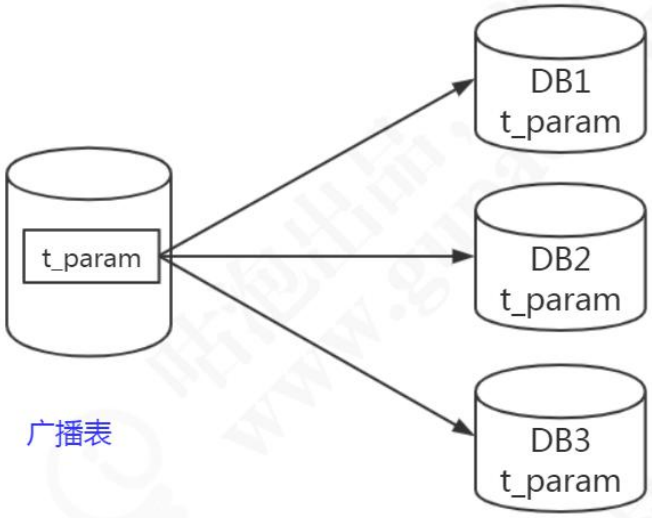
分片键不一定是主键，也不一定有业务含义。

1.6.2 动态表



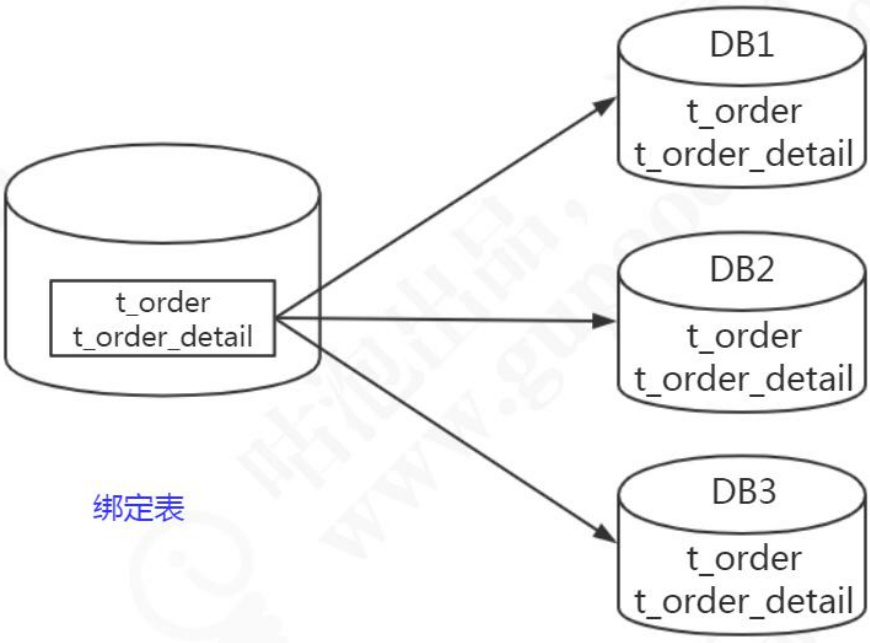
1.6.3 广播表

跟 Mycat 的全局表对应



1.6.4 绑定表

跟 Mycat 的 ER 表对应



1.7 使用规范

不支持的 SQL：

https://shardingsphere.apache.org/document/current/cn/features/sharding/use-norms/sql/

分页的说明：

https://shardingsphere.apache.org/document/current/cn/features/sharding/use-norms/pagination/

2 Sharding-JDBC 实战

快速入门

https://shardingsphere.apache.org/document/current/cn/quick-start/sharding-jdbc-quick-start/

2.1 引入依赖

注意，在 Spring Boot 中使用 Sharding-JDBC，可以直接引入 sharding-jdbc 的依赖。注意组织名称（groupId）的区别：

io.shardingjdbc ：更名之前

io.shardingsphere：更名之后

org.apache.shardingsphere：捐献给 Apache 之后

包名和某些类有差异，如果替换需要注意，import 的包名都需要修改。

核心依赖是（artifactId）：sharding-jdbc-core 和 sharding-core

前面两个 group 在 Spring Boot 中还提供了 starter，Apache 暂时没有。

2.2 原生 JDBC 使用

回顾：JDBC

gupao-shard-prop 工程 com.gupaoedu.jdbc.JdbcTest

gupao-shard-yml 工程 com.gupaoedu.jdbc.ShardJDBCTest

```
public class ShardJDBCTest {
  public static void main(String[] args) throws SQLException {
```

```
        // 配置真实数据源
        Map<String, DataSource> dataSourceMap = new HashMap<>();

        // 配置第一个数据源
        BasicDataSource dataSource1 = new BasicDataSource();
        dataSource1.setDriverClassName("com.mysql.jdbc.Driver");
        dataSource1.setUrl("jdbc:mysql://localhost:3306/shard0");
        dataSource1.setUsername("root");
        dataSource1.setPassword("123456");
        dataSourceMap.put("ds0", dataSource1);

        // 配置第二个数据源
        BasicDataSource dataSource2 = new BasicDataSource();
        dataSource2.setDriverClassName("com.mysql.jdbc.Driver");
        dataSource2.setUrl("jdbc:mysql://localhost:3306/shard1");
        dataSource2.setUsername("root");
        dataSource2.setPassword("123456");
        dataSourceMap.put("ds1", dataSource2);

        // 配置 Order 表规则
        TableRuleConfiguration orderTableRuleConfig = new TableRuleConfiguration();
        orderTableRuleConfig.setLogicTable("order");
        orderTableRuleConfig.setActualDataNodes("ds${0..1}.order${0..1}");


        // 配置分库 + 分表策略
        orderTableRuleConfig.setDatabaseShardingStrategyConfig(new InlineShardingStrategyConfiguration("orderId",
"ds${orderId % 2}"));
        orderTableRuleConfig.setTableShardingStrategyConfig(new InlineShardingStrategyConfiguration("orderId",
"order${orderId % 2}"));

        // 配置分片规则
        ShardingRuleConfiguration shardingRuleConfig = new ShardingRuleConfiguration();
        shardingRuleConfig.getTableRuleConfigs().add(orderTableRuleConfig);

        Map<String,Object> map = new HashMap<>();

        // 获取数据源对象
        DataSource dataSource = ShardingDataSourceFactory.createDataSource(dataSourceMap, shardingRuleConfig,map,
new Properties());

        String sql = "SELECT * from order WHERE orderId=?";
        try (
                Connection conn = dataSource.getConnection();
```

```
            PreparedStatement preparedStatement = conn.prepareStatement(sql)) {
            preparedStatement.setInt(1, 1);
        System.out.println();
            try (ResultSet rs = preparedStatement.executeQuery()) {
                while(rs.next()) {
                    // %2 结果，路由到 shard1.order1
                    System.out.println("orderId---------"+rs.getInt(1));
                    System.out.println("userId---------"+rs.getInt(2));
                    System.out.println("createTime---------"+rs.getInt(3));
                    System.out.println("totalPrice---------"+rs.getInt(4));
                }
            }
        }
    }
}
```

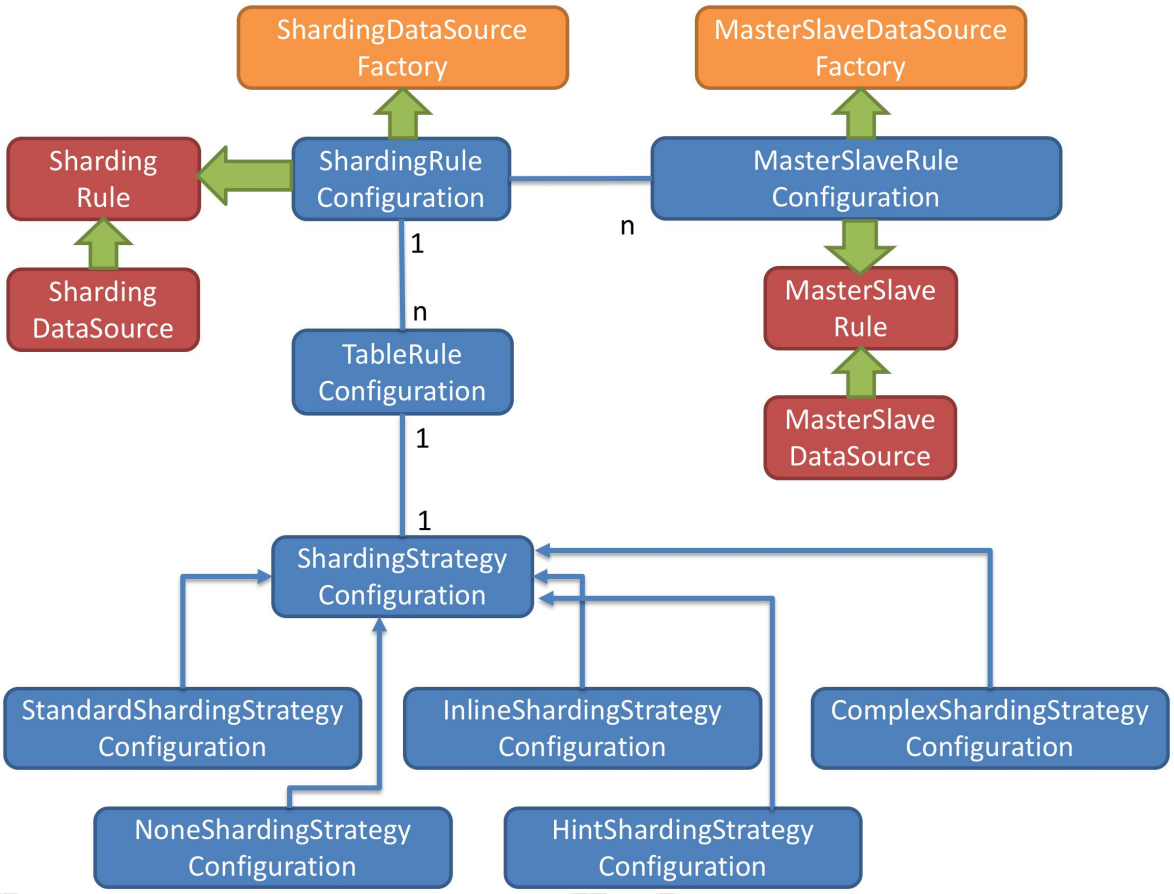
总结：ShardingRuleConfiguration 可以包含多个 TableRuleConfiguration（多张表），也可以设置默认的分库和分表策略。

每个 TableRuleConfiguration 可以针对表设置 ShardingStrategyConfiguration，包括分库分分表策略。

ShardingStrategyConfiguration 有 5 种实现（标准、行内、复合、Hint、无）。

ShardingDataSourceFactory 利用 ShardingRuleConfiguration 创建数据源。

有了数据源，就可以走 JDBC 的流程了。



在 JDBC 中使用，我们可以直接创建数据源，如果在 Spring 中使用，我们自定义的数据源怎么定义使用呢？可以通过注解或者 xml 配置文件注入。

2.3 Spring 中使用

先来总结一下，因为我们要使用 Sharding-JDBC 去访问数据库，所以我们不再使用 ORM 框架或者容器去定义数据源，而是注入 Sharding-JDBC 自定义的数据源，这样才能保证动态选择数据源的实现。

第二个，因为 Sharding-JDBC 是工作在客户端的，所以我们要在客户端配置分库分表的策略。跟 Mycat 不一样的是，Sharding-JDBC 没有内置各种分片策略和算法，需要我们通过表达式或者自定义的配置文件实现。我们创建的数据源中包含了分片的策略。

总体上，需要配置的就是这两个，数据源和分片策略，当然分片策略又包括分库的

策略和分表的策略。

配置的方式是多种多样的。

https://shardingsphere.apache.org/document/current/cn/manual/sharding-jdbc/configuration/config-java/

位置：4.用户手册——4.1 Sharding-JDBC——4.1.2 配置手册

2.3.1 Java 配置

gupao-shard-java 工程

第一种是把数据源和分片策略都写在 Java Config 中，它的特点是非常灵活，我们可以实现各种定义的分片策略。但是缺点是，如果把数据源、策略都配置在 Java Config 中，就出现了硬编码，在修改的时候比较麻烦。

2.3.2 Spring Boot 配置

gupao-shard-prop 工程

第二种是直接使用 Spring Boot 的 application.properties 来配置，这个要基于 starter 模块，org.apache.shardingsphere 的包还没有 starter，只有 io.shardingsphere 的包有 starter。

把数据源和分库分表策略都配置在 properties 文件中。这种方式配置比较简单，但是不能实现复杂的分片策略，不够灵活。

2.3.3 yml 配置

gupao-shard-yml 工程

第三种是使用 Spring Boot 的 yml 配置（shardingjdbc.yml），也要依赖 starter 模块。当然我们也可以结合不同的配置方式，比如把分片策略放在 JavaConfig 中，数据源配置在 yml 中或 properties 中。

2.4 Spring 案例验证（gupao-shard-prop 工程）

我们这里验证的是切分到本地的两个库 ds0，ds1。

注意：之前 Mycat 的课程演示，不同的数据节点都在不同的机器上，这里我们以同一数据库服务中不同的 database 来替代。无论是多个 IP 的多个库还是一个 IP 的多个库，对于验证来说没有区别。

两个库里面都是相同的 4 张表（user_info，t_order，t_order_item，t_config），（sys_user 是 AbstractRoutingDataSource 用的，不用管）。

这些表必须提前创建，中间件是不会帮我们生成表结构的。

然后我们用 MyBatis 的 Generator 生成相应的实体类、Mapper 接口和映射器。

对数据库的基本的 SSM 的操作弄完了，接下来就是分库分表的配置，一个是数据源，一个是分片策略。

我们先来看一下我们的数据源的配置 application.properties。

```
sharding.jdbc.datasource
```

如果用 Spring 管理数据源：

```
spring.datasource.url=jdbc:mysql://192.168.8.168:8066/gupao
spring.datasource.username=root
spring.datasource.password=123456
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

当我们使用了 Sharding-JDBC 的数据源以后，对于数据的操作会交给 Sharding-JDBC 的代码来处理。

分片策略从维度上分成两种，一种是分库，一种是分表。

我们可以定义默认的分库分表策略（配置中注释了），例如：用 user_id 作为分片键。

这里用到了一种分片策略的实现，叫做行内表达式。我们对 user_id 取模，然后选择数据库。如果模 2 等于 0，在第一个数据库中。模 2 等于 1，在第二个数据库中。

数据源名称是行内表达式组装出来的。

```
sharding.jdbc.config.sharding.default-database-strategy.inline.sharding-column=user_id
sharding.jdbc.config.sharding.default-database-strategy.inline.algorithm-expression=ds${user_id % 2}
```

对于不同的表，也可以单独配置分库策略（databaseStrategy）和分表策略（tableStrategy）。案例中只有分库没有分表，所以没定义 tableStrategy。

使用以下配置打印路由信息（或放在 yml 中）：

```
sharding.jdbc.config.sharding.props.sql.show=true
```

2.4.1 取模分片

我们用 user_info 表来验证取模分片。根据 user_id，把用户数据划分到两个数据节点上。

在本地创建两个数据库 ds0 和 ds1，都创建 user_info 表：

```
CREATE TABLE `user_info` (
`user_id` bigint(19) NOT NULL,
`user_name` varchar(45) DEFAULT NULL,
`account` varchar(45) NOT NULL,
`password` varchar(45) DEFAULT NULL,
PRIMARY KEY (`user_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

这里只定义了分库策略，没有定义单库内的分表策略，两个库都是相同的表名。

路由的结果：ds0.user_info，ds1.user_info。

如果定义了分库策略，两个库里面都有两张表，那么路由的结果可能是 4 种：

ds0.user_info0，ds0.user_info1；

ds1. user_info0, ds1. user_info1

```
sharding.jdbc.config.sharding.tables.user_info.databaseStrategy.inline.shardingColumn=user_id
sharding.jdbc.config.sharding.tables.user_info.databaseStrategy.inline.algorithm-expression=ds${user_id % 2}
#sharding.jdbc.config.sharding.tables.user_info.table-strategy.inline.sharding-column=user_id
#sharding.jdbc.config.sharding.tables.user_info.table-strategy.inline.algorithm-expression=user_info
sharding.jdbc.config.sharding.tables.user_info.actual-data-nodes=ds$->{0..1}.user_info
```

gupao-shard-prop 工程：

在我们的单元测量测试类 UserShardingTest 里面，执行 insert()，调用 Mapper 接口循环插入 100 条数据。

我们看一下插入的结果。user_id 为偶数的数据，都落到了第一个库。user_id 为奇数的数据，都落到了第二个库。

执行 select()测一下查询，看看数据分布在两个节点的时候，我们用程序查询，能不能取回正确的数据。

2.4.2 绑定表

第二种是绑定表，也就是父表和子表有关联关系。主表和子表使用相同的分片策略。

```
CREATE TABLE `t_order` (
`order_id` int(11) NOT NULL,
`user_id` int(11) NOT NULL,
PRIMARY KEY (`order_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE `t_order_item` (
`item_id` int(11) NOT NULL,
`order_id` int(11) NOT NULL,
`user_id` int(11) NOT NULL,
```

```
PRIMARY KEY (`item_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

除了定义分库和分表算法之外，我们还需要多定义一个 binding-tables。

绑定表将使用主表的分片策略。

```
sharding.jdbc.config.sharding.tables.user_info.databaseStrategy.inline.shardingColumn=user_id
sharding.jdbc.config.sharding.tables.user_info.databaseStrategy.inline.algorithm-expression=ds${user_id % 2}
#sharding.jdbc.config.sharding.tables.user_info.table-strategy.inline.sharding-column=user_id
#sharding.jdbc.config.sharding.tables.user_info.table-strategy.inline.algorithm-expression=user_info
sharding.jdbc.config.sharding.tables.user_info.actual-data-nodes=ds$->{0..1}.user_info

# 分库算法 t_order 多库分表
sharding.jdbc.config.sharding.tables.t_order.databaseStrategy.inline.shardingColumn=order_id
sharding.jdbc.config.sharding.tables.t_order.databaseStrategy.inline.algorithm-expression=ds${order_id % 2}
sharding.jdbc.config.sharding.tables.t_order.actual-data-nodes=ds$->{0..1}.t_order

# 分库算法 t_order_item 多库分表
sharding.jdbc.config.sharding.tables.t_order_item.databaseStrategy.inline.shardingColumn=order_id
sharding.jdbc.config.sharding.tables.t_order_item.databaseStrategy.inline.algorithm-expression=ds${order_id % 2}
sharding.jdbc.config.sharding.tables.t_order_item.actual-data-nodes=ds$->{0..1}.t_order_item

# 绑定表规则列表
sharding.jdbc.config.sharding.binding-tables[0]=t_order,t_order_item
```

绑定表不使用分片键查询时，会出现笛卡尔积。

什么叫笛卡尔积？假如有 2 个数据库，两张表要相互关联，两张表又各有分表，那么 SQL 的执行路径就是 2*2*2=8 种。

gupao-shard-yml 工程：com.gupaoedu.OrderTest#selectOrderItem

```
: Actual SQL: ds0 ::: select a.* from order_item1 a inner join order0 b on
: Actual SQL: ds0 ::: select a.* from order_item1 a inner join order1 b on
: Actual SQL: ds0 ::: select a.* from order_item0 a inner join order0 b on
: Actual SQL: ds0 ::: select a.* from order_item0 a inner join order1 b on
: Actual SQL: ds1 ::: select a.* from order_item1 a inner join order0 b on
: Actual SQL: ds1 ::: select a.* from order_item1 a inner join order1 b on
: Actual SQL: ds1 ::: select a.* from order_item0 a inner join order0 b on
: Actual SQL: ds1 ::: select a.* from order_item0 a inner join order1 b on
```

使用分片键 com.gupaoedu.OrderTest#joinById()

```
Actual SQL: ds0 ::: select a.* from order_item1 a inner join order1 b on
Actual SQL: ds0 ::: select a.* from order_item0 a inner join order1 b on
Actual SQL: ds1 ::: select a.* from order_item1 a inner join order1 b on
Actual SQL: ds1 ::: select a.* from order_item0 a inner join order1 b on
```

mycat 不支持这种二维的路由。

我们去看一下测试的代码 OrderShardingTest 和 OrderItemShardingTest。

先插入主表的数据，再插入子表的数据。再查询一下看分片是否正确。

2.4.3 广播表

最后一种是广播表，也就是需要在所有节点上同步操作的数据表。

```
# 广播表
sharding.jdbc.config.sharding.broadcast-tables=t_config
```

我们用 broadcast-tables 来定义。

gupao-shard-prop 工程：ConfigShardingTest.java

插入和更新都会在所有节点上执行。

如果我们需要更加复杂的分片策略，properties 文件中行内表达式的这种方式肯定

满足不了。实际上 properties 里面的分片策略可以指定，比如 user_info 表的分库和分表策略。

```
sharding.jdbc.config.sharding.tables.user_info.tableStrategy.standard.shardingColumn=
sharding.jdbc.config.sharding.tables.user_info.tableStrategy.standard.preciseAlgorithmClassName=
sharding.jdbc.config.sharding.tables.user_info.tableStrategy.standard.rangeAlgorithmClassName=
```

这个时候我们需要了解 Sharding-JDBC 中几种不同的分片策略。

3 分片策略详解（gupao-shard-java）

https://shardingsphere.apache.org/document/current/cn/features/sharding/concept/sharding/

Sharding-JDBC 中的分片策略有两个维度：分库（数据源分片）策略和分表策略。

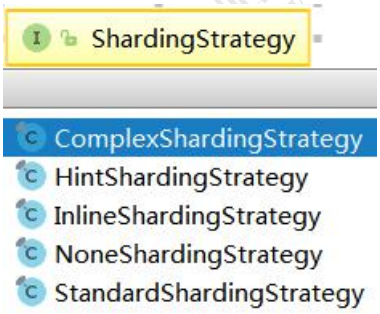
分库策略表示数据路由到的物理目标数据源，分表分片策略表示数据被路由到的目标表。分表策略是依赖于分库策略的，也就是说要先分库再分表，当然也可以不分库只分表。

跟 Mycat 不一样 ,Sharding-JDBC 没有提供内置的分片算法 ,而是通过抽象成接口，让开发者自行实现，这样可以根据业务实际情况灵活地实现分片。

3.1 分片策略

包含分片键和分片算法，分片算法是需要自定义的。可以用于分库，也可以用于分表。

Sharding-JDBC 提供了 5 种分片策略，这些策略全部继承自 ShardingStrategy。



3.1.1 行表达式分片策略

https://shardingsphere.apache.org/document/current/cn/features/sharding/other-features/inline-expression/

对应 InlineShardingStrategy 类。只支持单分片键，提供对=和 IN 操作的支持。行内表达式的配置比较简单。

例如：

${begin..end}表示范围区间

${[unit1, unit2, unit_x]}表示枚举值

t_user_$->{u_id % 8} 表示 t_user 表根据 u_id 模 8，而分成 8 张表，表名称为 t_user_0 到 t_user_7。

行表达式中如果出现连续多个${ expression }或$->{ expression }表达式，整个表达式最终的结果将会根据每个子表达式的结果进行笛卡尔组合。

例如，以下行表达式：

${['db1', 'db2']}_table${1..3}

最终会解析为：

db1_table1, db1_table2, db1_table3,

db2_table1, db2_table2, db2_table3

3.1.2 标准分片策略

对应 StandardShardingStrategy 类。

标准分片策略只支持单分片键，提供了提供 PreciseShardingAlgorithm 和 RangeShardingAlgorithm 两个分片算法，分别对应于 SQL 语句中的=, IN 和 BETWEEN

AND。

如果要使用标准分片策略，必须要实现 PreciseShardingAlgorithm，用来处理=和 IN 的分片。RangeShardingAlgorithm 是可选的。如果没有实现，SQL 语句会发到所有的数据节点上执行。

3.1.3 复合分片策略

比如：根据日期和 ID 两个字段分片，每个月 3 张表，先根据日期，再根据 ID 取模。

对应 ComplexShardingStrategy 类。可以支持等值查询和范围查询。

复合分片策略支持多分片键，提供了 ComplexKeysShardingAlgorithm，分片算法需要自己实现。

3.1.4 Hint 分片策略

对应 HintShardingStrategy。通过 Hint 而非 SQL 解析的方式分片的策略。有点类似于 Mycat 的指定分片注解。

https://shardingsphere.apache.org/document/current/cn/manual/sharding-jdbc/usage/hint/

3.1.5 不分片策略

对应 NoneShardingStrategy。不分片的策略。

3.2 分片算法

创建了分片策略之后，需要进一步实现分片算法。Sharding-JDBC 目前提供 4 种分片算法。

3.2.1 精确分片算法

对应 PreciseShardingAlgorithm，用于处理使用单一键作为分片键的=与 IN 进行

分片的场景。需要配合 StandardShardingStrategy 使用。

3.2.2 范围分片算法

对应 RangeShardingAlgorithm，用于处理使用单一键作为分片键的 BETWEEN AND 进行分片的场景。需要配合 StandardShardingStrategy 使用。

如果不配置范围分片算法，范围查询默认会路由到所有节点。

3.2.3 复合分片算法

对应 ComplexKeysShardingAlgorithm ,用于处理使用多键作为分片键进行分片的场景，包含多个分片键的逻辑较复杂，需要应用开发者自行处理其中的复杂度。需要配合 ComplexShardingStrategy 使用。

3.2.4 Hint 分片算法

对应 HintShardingAlgorithm，用于处理使用 Hint 行分片的场景。需要配合 HintShardingStrategy 使用。

https://shardingsphere.apache.org/document/current/cn/manual/sharding-jdbc/usage/hint/

3.2.5 算法实现

gupao-shard-java

所有的算法都需要实现对应的接口，实现 doSharding()方法：

例如：PreciseShardingAlgorithm

传入分片键，返回一个精确的分片（数据源名称）

```
String doSharding(Collection<String> availableTargetNames, PreciseShardingValue<T> shardingValue);
```

RangeShardingAlgorithm

传入分片键，返回多个数据源名称

```
Collection<String> doSharding(Collection<String> availableTargetNames, RangeShardingValue<T> shardingValue);
```

ComplexKeysShardingAlgorithm

传入多个分片键，返回多个数据源名称

```
Collection<String> doSharding(Collection<String> availableTargetNames, Collection<ShardingValue> shardingValues);
```

4 分布式事务（shard-gupao-yml）

4.1 事务概述

https://shardingsphere.apache.org/document/current/cn/features/transaction/

为什么要有分布式事务？

XA 模型的不足：需要锁定资源

柔性事务

4.2 两阶段事务-XA

gupao-shard-yml

com.gupaoedu.TransactionTest

```
<dependency>
  <groupId>io.shardingsphere</groupId>
  <artifactId>sharding-transaction-2pc-xa</artifactId>
  <version>3.1.0</version>
</dependency>
```

默认是用 atomikos 实现的。

在 Service 类上加上注解

```
@ShardingTransactionType(TransactionType.XA)
@Transactional(rollbackFor = Exception.class)
```

其他事务类型：Local、BASE

模拟在两个节点上操作，id=12673、id=12674 路由到两个节点，第二个节点插入两个相同对象，发生主键冲突异常，会发生回滚。

XA 实现类：

XAShardingTransactionManager——XATransactionManager——AtomikosTransactionManager

4.3 柔性事务 Saga

ShardingSphere 的柔性事务已通过第三方 SPI 实现 Saga 事务，Saga 引擎使用 Servicecomb-Saga。

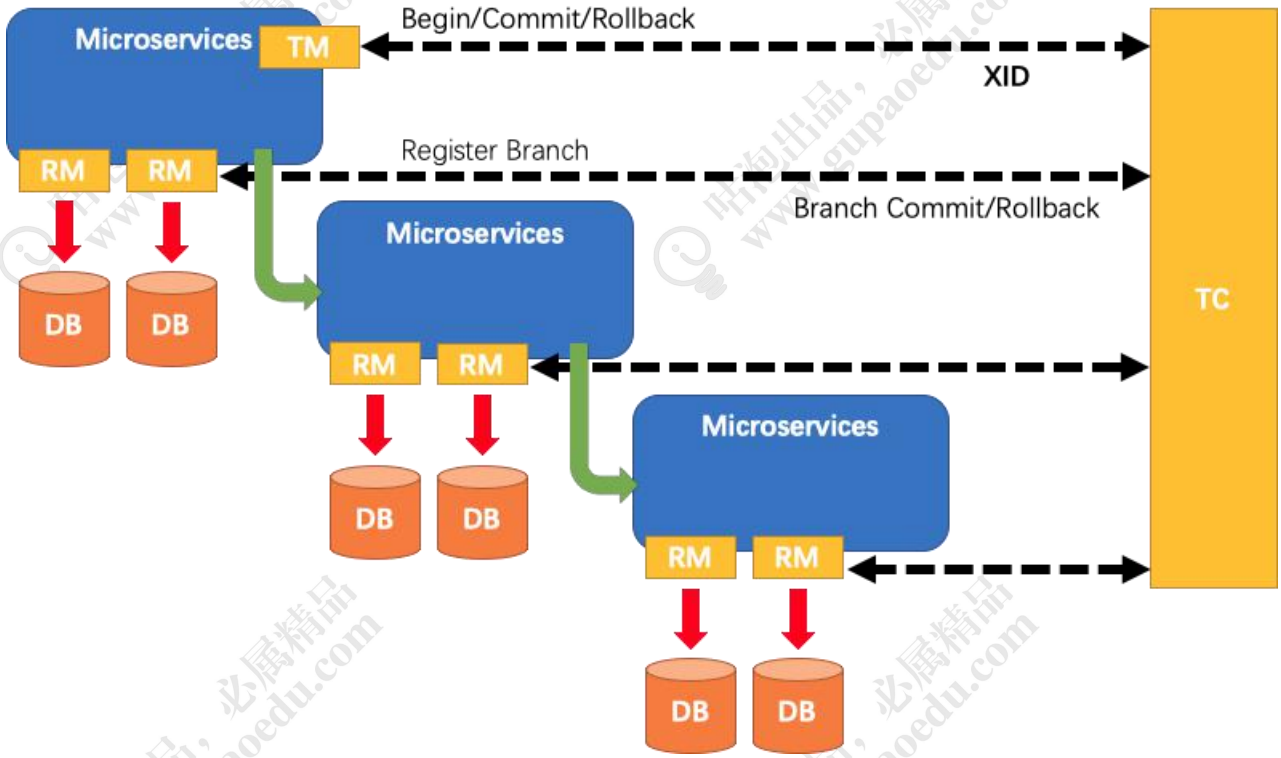
参考官方的这篇文章《分布式事务在 Sharding-Sphere 中的实现》

```
<dependency>
  <groupId>io.shardingsphere</groupId>
  <artifactId>sharding-transaction-2pc-spi</artifactId>
  <version>3.1.0</version>
</dependency>
```

4.4 柔性事务 Seata

https://github.com/seata/seata
https://github.com/seata/seata-workshop
https://mp.weixin.qq.com/s/xfUGep5XMcIqRTGY3WFpgA

GTS 的社区版本叫 Fescar（Fast & Easy Commit And Rollback），Fescar 改名后叫 Seata AT（Simple Extensible Autonomous Transaction Architecture）。



需要额外部署 Seata-server 服务进行分支事务的协调。

官方的 demo 中有一个例子：

https://github.com/apache/incubator-shardingsphere-example

incubator-shardingsphere-example-dev\sharding-jdbc-example\transaction-example\transaction-base-seata-raw-jdbc-example

5 分布式全局 ID（shard-gupao-yml）

https://shardingsphere.apache.org/document/current/cn/features/sharding/other-features/key-generator/

在 shard-gupao-yml 中，使用 key-generator-column-name 配置，生成了一个 18 位的 ID。

Properties 配置：

```
sharding.jdbc.config.sharding.default-key-generator-class-name=
sharding.jdbc.config.sharding.tables.t_order.keyGeneratorColumnName=
sharding.jdbc.config.sharding.tables.t_order.keyGeneratorClassName=
```

Java config 配置：

```
tableRuleConfig.setKeyGeneratorColumnName("order_id");
tableRuleConfig.setKeyGeneratorClass("io.shardingsphere.core.keygen.DefaultKeyGenerator");
```

keyGeneratorColumnName：指定需要生成 ID 的列

KeyGenerotorClass：指定生成器类，默认是 DefaultKeyGenerator.java，里面使用了雪花算法。

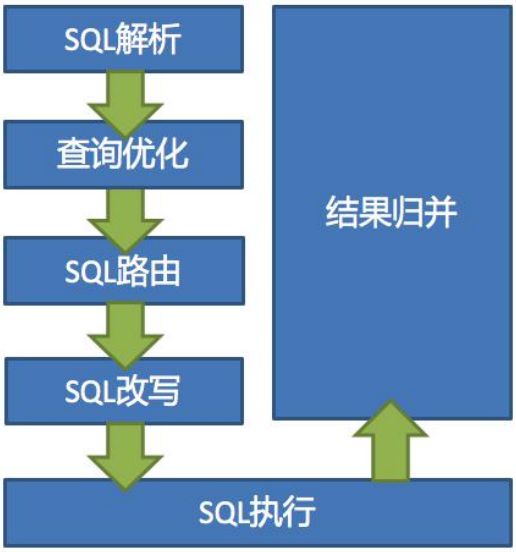
6 Sharding-JDBC 工作流程

内核剖析

https://shardingsphere.apache.org/document/current/cn/features/sharding/principle/

Sharding-JDBC 的原理总结起来很简单：

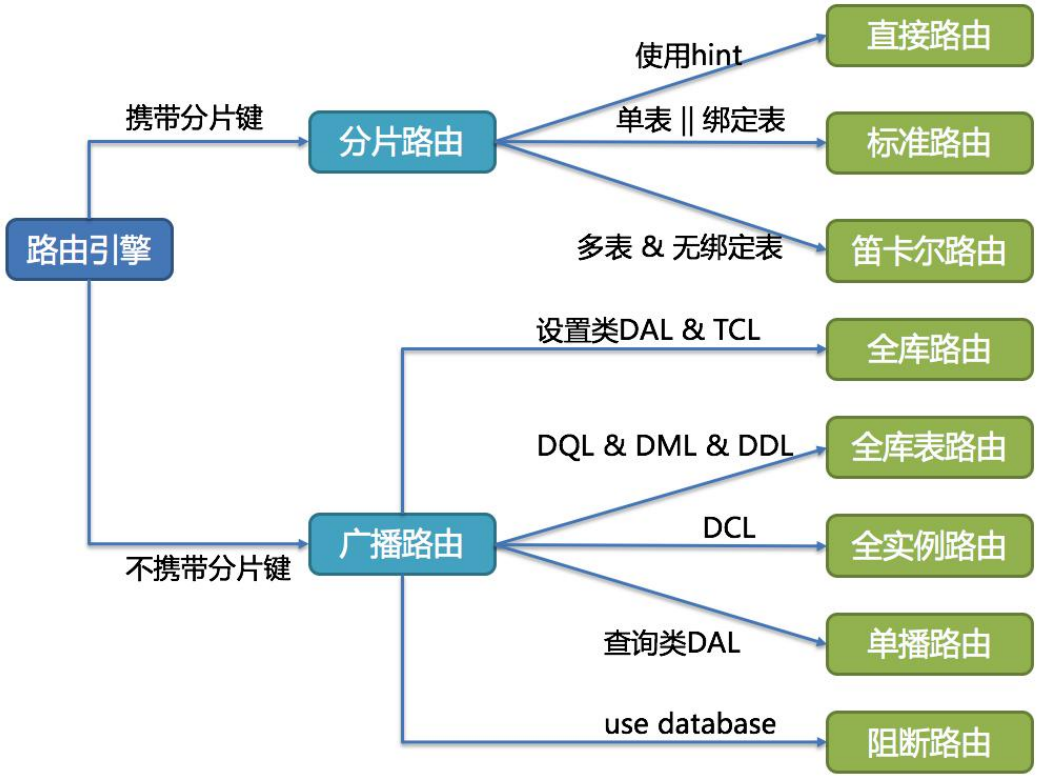
SQL 解析 => 执行器优化 => SQL 路由 => SQL 改写 => SQL 执行 => 结果归并。



6.1 SQL 解析

SQL 解析主要是词法和语法的解析。目前常见的 SQL 解析器主要有 fdb，jsqlparser 和 Druid。Sharding-JDBC1.4.x 之前的版本使用 Druid 作为 SQL 解析器。从 1.5.x 版本开始，Sharding-JDBC 采用完全自研的 SQL 解析引擎。

6.2 SQL 路由



SQL 路由是根据分片规则配置以及解析上下文中的分片条件，将 SQL 定位至真正的数据源。它又分为直接路由、简单路由和笛卡尔积路由。

直接路由，使用 Hint 方式。

Binding 表是指使用同样的分片键和分片规则的一组表，也就是说任何情况下，Binding 表的分片结果应与主表一致。例如：order 表和 order_item 表，都根据 order_id 分片，结果应是 order_1 与 order_item_1 成对出现。这样的关联查询和单表查询复杂度

和性能相当。如果分片条件不是等于，而是 BETWEEN 或 IN，则路由结果不一定落入单库（表），因此一条逻辑 SQL 最终可能拆分为多条 SQL 语句。

笛卡尔积查询最为复杂，因为无法根据 Binding 关系定位分片规则的一致性，所以非 Binding 表的关联查询需要拆解为笛卡尔积组合执行。查询性能较低，而且数据库连接数较高，需谨慎使用。

6.3 SQL 改写

例如：将逻辑表名称改成真实表名称，优化分页查询等

6.4 SQL 执行

因为可能链接到多个真实数据源， Sharding -JDBC 将采用多线程并发执行 SQL。

6.5 结果归并

例如数据的组装、分页、排序等等。

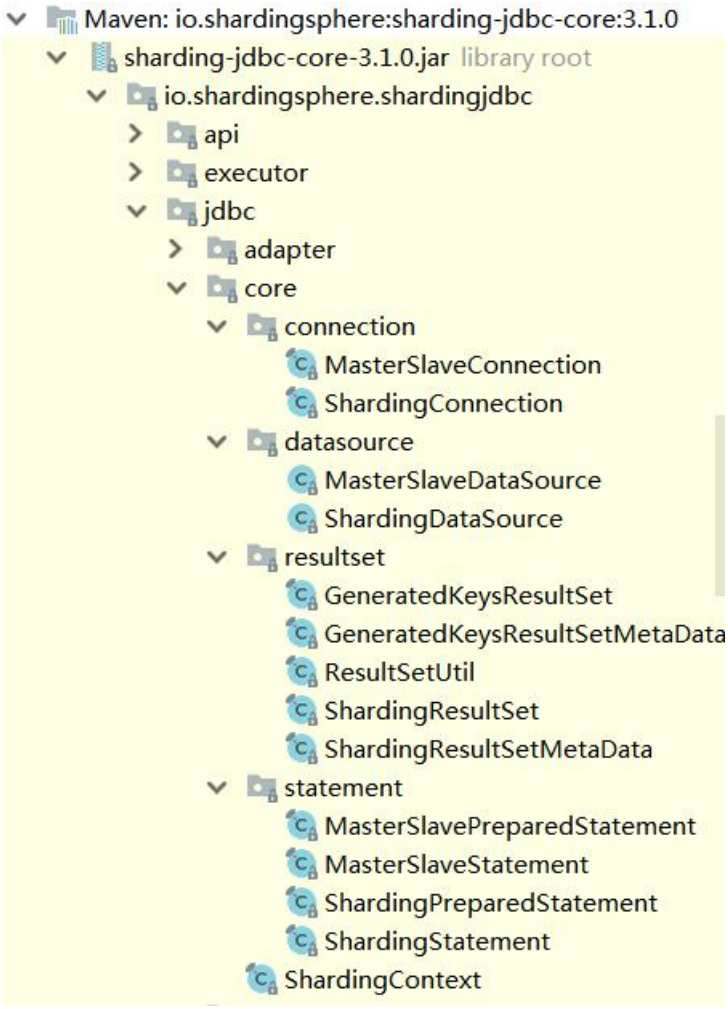
7 Sharding-JDBC 实现原理

大家还记不记得，我们在 Mycat 的第一节课说到 JDBC 的四大核心对象？

DataSource、Connection、Statement（PS）、ResulstSet。

Sharding-JDBC 封装了这四个核心类，在类名前面加上了 Sharding。

gupao-shard-prop 工程



如果说带 Sharding 的类要替换 JDBC 的对象，那么一定要找到创建和调用他们的地方。ShardingDataSource 我们不说了，系统启动的时候就创建好了。

问题就在于，我们是什么时候用 ShardingDataSource 获取一个 ShardingConnection 的？

我们以整合了 MyBatis 的项目为例。MyBatis 封装了 JDBC 的核心对象，那么在 MyBatis 操作 JDBC 四大对象的时候，就要替换成 Sharding-JDBC 的四大对象。

没有看过 MyBatis 源码的同学一定要去看看。我们的查询方法最终会走到 SimpleExecutor 的 doQuery()方法，这个是我们的前提知识，那我们直接在 doQuery()打断点。

doQuery()方法里面调用了 prepareStatement()创建连接

```
private Statement prepareStatement(StatementHandler handler, Log statementLog) throws SQLException {
  Statement stmt;
  Connection connection = getConnection(statementLog);
  stmt = handler.prepare(connection, transaction.getTimeout());
  handler.parameterize(stmt);
  return stmt;
}
```

它经过以下两个方法，返回了一个 ShardingConnection。

DataSourceUtil.fetchConnection()

Connection con = dataSource.getConnection();

基于这个 ShardingConnection，最终得到一个 ShardingStatement

```
stmt = handler.prepare(connection, transaction.getTimeout());
```

接下来就是执行

```
return handler.query(stmt, resultHandler);
```

再调用了的 ShardingStatement 的 execute()

```
public <E> List<E> query(Statement statement, ResultHandler resultHandler) throws SQLException {
  PreparedStatement ps = (PreparedStatement) statement;
  ps.execute();
  return resultSetHandler.handleResultSets(ps);
}
```

最终调用的是 ShardingPreparedStatement 的 execute 方法。

```
public boolean execute() throws SQLException {
  try {
      clearPrevious();
      sqlRoute();
      initPreparedStatementExecutor();
      return preparedStatementExecutor.execute();
  } finally {
```

```
        refreshTableMetaData(connection.getShardingContext(), routeResult.getSqlStatement());
        clearBatch();
    }
}
```

SQL 的解析路由就是在这一步完成的。

8 Sharding-Proxy 介绍

下载地址：

https://github.com/sharding-sphere/sharding-sphere-doc/raw/master/dist/sharding-proxy-3.0.0.tar.gz

lib 目录就是 sharding-proxy 核心代码，以及依赖的 JAR 包；

bin 目录就是存放启停脚本的；

conf 目录就是存放所有配置文件，包括 sharding-proxy 服务的配置文件、数据源以及 sharding 规则配置文件和项目日志配置文件。

Linux 运行 start.sh 启动（windows 用 start.bat），默认端口 3307

需要的自定义分表算法，只需要将它编译成 class 文件，然后放到 conf 目录下，也可以打成 jar 包放在 lib 目录下。

9 与 Mycat 对比

	Sharding-JDBC	**Mycat**
工作层面	JDBC 协议	MySQL 协议/JDBC 协议
运行方式	Jar 包，客户端	独立服务，服务端
开发方式	代码/配置改动	连接地址（数据源）
运维方式	无	管理独立服务，运维成本高
性能	多线程并发按操作，性能高	独立服务+网络开销，存在性能损失风险
功能范围	协议层面	包括分布式事务、数据迁移等
适用操作	OLTP	OLTP+OLAP
支持数据库	基于 JDBC 协议的数据库	MySQL 和其他支持 JDBC 协议的数据库
支持语言	Java 项目中使用	支持 JDBC 协议的语言

从易用性和功能完善的角度来说，Mycat 似乎比 Sharding-JDBC 要好，因为有现成的分片规则，也提供了 4 种 ID 生成方式，通过注解可以支持高级功能，比如跨库关联查询。

建议：小型项目，分片规则简单的项目可以用 Sharding-JDBC。大型项目，可以用 Mycat。